PERFORMANCE EVALUATION OF INTERRUPT HANDLING SCHEMES IN GIGABIT NETWORKS

by

Khalid Abdalla El-Badawi

A Thesis Presented to the

DEANSHIP OF GRADUATE STUDIES

In Partial Fulfillment of the Requirements

for the degree

MASTER OF SCIENCE

IN

COMPUTER SCIENCE

KING FAHD UNIVERSITY

OF PETROLEUM & MINERALS

Dhahran, Saudi Arabia

April, 2003

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN 31261, SUADI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis written by **KHALID ABDALLA EL-BADAWI** under the direction of his thesis advisor and approved by his thesis committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN COMPUTER SCIENCE**.

Thesis Committee

_____
Dr. Khalid Salah (Thesis Advisor)

_____
Dr. Nasir Al-Darwish (Member)

_____
Dr. Muhammed Alsuwaiyel (Member)

_____
Department Chairman
Dr. Kanaan Faisal

_____
Dean of Graduate Studies
Prof. Osama A. Jannadi

_____
Date

Dedication

This thesis is lovingly dedicated to my mother

Mrs. Fatima Mustafa

for all I am started in her arms.

Acknowledgment

All thanks are due Allah first and foremost for his countless blessing. Acknowledgment is due to King Fahd University of Petroleum & Minerals for supporting this research.

My unrestrained appreciation goes to my advisor, Dr. Khalid Salah, for all the assistance, advice, encouragement and invaluable support he has given me throughout the course of this work and on several other occasions. I simply cannot begin to imagine how things would have proceeded without his help and his patience. I also wish to thank my thesis committee members, Dr. Nasir Darwish and Dr. Muhammed Alsuwaiyel, for their help, support, and contributions. I would also like to thank Dr. Haydar Akca for his valuable help.

I also acknowledge my many colleagues and friends as I had a pleasant, enjoyable and fruitful company with them. Specially, I would like to thank my friend Abdel-Rahman for his continuous encouragement and support.

Finally, I wish to express my gratitude to my family members for being patient with me and offering words of encouragements to spur my spirit at moments of depression.

Table of Contents

List of Figures

Thesis Abstract

NAME: Khalid Abdalla El-Badawi

TITLE: Performance Evaluation of Interrupt Handling Schemes in Gigabit Networks.

MAJOR FIELD: Computer Science.

DATE OF DEGREE: April 2003.

In Gigabit networks, the arrival rate of incoming traffic is very high and supercedes the packet processing rate of network nodes such as router, servers, or clients. In addition, the very high rate of incoming traffic causes a very high rate of interrupts which has negative impact on the operating system performance of these network nodes. The negative impact is primarily due to interrupt overhead associated with each packet arrival. This thesis presents models and analytical techniques for capturing the behavior and studying the performance of interrupt-driven kernels due to Gigabit networks traffic. The Performance is expressed in terms of throughput, latency, CPU availability, and overall power system. In addition, the thesis evaluates and compares the performance of four popular interrupt handling schemes for decreasing such interrupt overhead. These schemes include Traditional scheme, Interrupt Coalescing, Polling, and Enabling and Disabling Interrupt. The performance for all of these schemes is studied using both analysis and simulation. Finally, the thesis discusses important selection, design, and implementation issues as well proposing the selection for the best interrupt handling scheme.

خلاصة الرسالة

الإســـــم : خالد عبدالله البدوي

عنوان الرسالة : تقييم أداء طرق معالجة المقاطعات (interrupts) في شبكات الجيجابت

التخصص : علوم الحاسب الآلي

تاريخ التخرج : أبريل ٢٠٠٣

في شبكات الجيجابت ، إن معدل وصول حركة سير القدوم تكون عالية جداً وقد تفوق معدل معالجة الحـزم عنـد طرفيات الشبكة مثل الخادمات و الراوتر وأجهزة العملاء. علاوة على ذلك ، قد يسبب هذا المعدل العـالي لحركـة السير تولد مقاطعات بمعدل عالي جداً والتي لها أثر سلبي على أداء نظام التشغيل لتلك الطرفيات. هذا الأثر السـلبي ناتج أساساً من تكلفة المقاطعة المصاحبة مع كل حزمة قادمة. يقدم هذا البحث نماذج وطرق تحليلية لمعرفـة سـلوك ودراسة أداء نواة الأنظمة المنقادة بالمقاطعة (interrupt-driven kernels) تحت تـأثير حركـة سـير شـبكات الجيجابت. يمكن أن نعبر عن هذا الأداء بدلالة الإخراج (throughput) ، التأخر (latency) ، توفر وحدة المعالجة المركزية (CPU availability) و قوة النظام الكلي. إضافة إلى ذلك ، هذا البحث يقيم ويقارن أداء أربـع طـرق معالجة المقاطعة لتخفيض تكلفتها. تشمل هذه الطرق: الطريقة التقليدية وطريقة جمع المقاطعات وطريقة الانتخـاب و طريقة تمكين-عدم تمكين المقاطعات. تم دراسة أداء هذه الطرق باستخدام التحليل والمحاكاة. وأخيراً ، يناقش هـذا البحث عدة أمور منها أهمية اختيار الطريقة ، التصميم ، والتطبيق كما يقترح اختيار الطريقة المثلـى لطـرق معالجـة المقاطعة.

CHAPTER 1

INTRODUCTION

1.1 Gigabit Ethernet Technology

These days we have a widespread deployment and development of high-performance network services, which provide high bandwidth and low latency. One of such network services is Gigabit Ethernet which was introduced in 1998. Like Ethernet, Gigabit Ethernet is *media access control* (MAC) and *physical-layer* (PHY) technology. It offers one gigabit per second (1 Gbps) raw bandwidth. To remain backward compatible with existing Ethernet technologies, Gigabit Ethernet, also known as IEEE Standard 802.3z, uses the same IEEE 802.3 Ethernet frame format.

Like its predecessor, Gigabit Ethernet operates in either half-duplex or full-duplex mode. In full-duplex mode, frames travel in both directions simultaneously over two separate channels on the same connection for an aggregate bandwidth of twice that of half-duplex mode. Full duplex networks are very efficient since data can be sent and received simultaneously. However, full-duplex transmission, which is commonly implemented, can be used for point-to-point connections only.

Full-duplex transmission can be deployed between ports on two switches, a workstation and a switch port, or between two workstations. Full-duplex connections

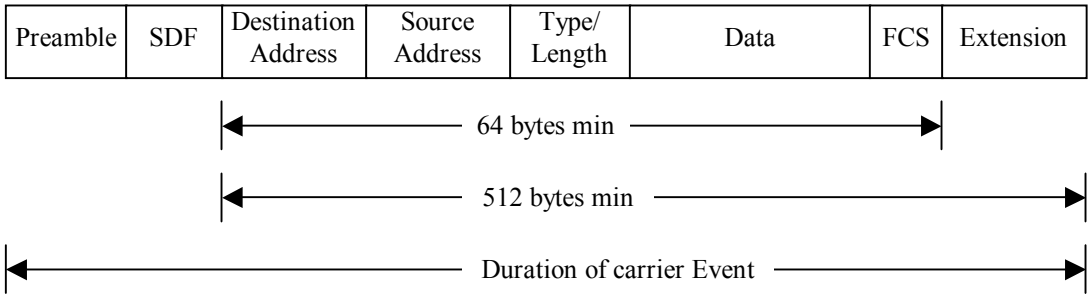
cannot be used for share-port connections, such as a repeater or hup port that connects multiple workstations. Gigabit Ethernet is most effective when running in the full-duplex, point-to-point mode where full bandwidth is dedicated between the two end-nodes. Full-duplex operation is ideal for backbones and high-speed server or router links.

For half-duplex operation, Gigabit Ethernet will use the enhanced CSMA/CD access method. With CSMA/CD, the same channel can only transmit or receive at one time. A collision results when a frame sent from one end of the network collides with another frame. Timing becomes critical if and when a collision occurs. If a collision occurs during the transmission of a frame, the MAC will stop transmitting and retransmit the frame when the transmission medium is clear. If the collision occurs after a packet has been sent, then the packet is lost since the MAC has already discarded the frame and started to prepare for the next frame for transmission. In all cases, the rest of the network must wait for the collision to dissipate before any other devices can transmit.

In half duplex mode, Gigabit Ethernet's performance is degraded. This is because Gigabit Ethernet uses CSMA/CD protocol which is sensitive to frame length. Ethernet has a minimum frame size of 64 bytes. The reason for having a minimum size is to prevent a station from completing the transmission of a frame before the first bit has reached the far end of the cable, where it may collide with another frame. Therefore, the minimum time to detect a collision is the time it takes for the signal to propagate from one end of the cable to the other. This minimum time is called *slot time*. The standard slot time for Ethernet frames is not long enough to run a 200-meter cable when

passing 64-byte frames at Gigabit speed. In order to accommodate the timing problem experienced with CSMA/CD when scaling half-duplex Ethernet to Gigabit speed, slot time has been extended to guarantee at least a 512-byte slot time using a technique called *carrier extension* as shown in Figure 1.1. The frame size is not changed; only the timing is extended.

Carrier Extension wastes bandwidth. For example, a small packet of 64 bytes will have 448 padding bytes of carrier extension symbols. This clearly results in low throughput and an increased collision rate which may increase the number of lost frames. In fact, for a large number of small packets, the Gigabit Ethernet throughput is only marginally better than 100BaseT.



SDF : Start of Frame Delimiter.
FCS: Frame Check Sequence.

Figure 1.1: Gigabit Ethernet frame format with carrier extension

To gain back some of the performance lost due to carrier extension, Nbase Communication (Chatsworth, California) proposed a solution known as *packet bursting*. It is essentially a modification to the carrier extension procedure. The idea is to transmit a burst of frames every time the first frame has successfully passed the collision window

of 512 bytes. Carrier extension is only applied to the first frame in a burst. This essentially averages the wasted time in the carrier extension symbols over the few frames that are transmitted. Packet bursting substantially increases the throughput and does not change the dynamics of the CSMA/CD algorithm. It only slightly modified the existing MAC definition.

Half-duplex operation is intended for shared multi-station LANs, where two or more end nodes share a single port. Most switches enable users to select half-duplex or full-duplex operation on a port-by-port basis, allowing users to migrate from shared links to point-to-point, full duplex links when they are ready.

1.2 Interrupt-Driven Kernels

Many applications such as video streaming and voice over IP impose heavy demands on the communication network. Gigabit Ethernet technology can provide the required performance to meet these demands. However, it has also shifted the communication bottleneck from network interconnections to host systems.

There are two main problems seen in Gigabit networking that reduce the performance of host systems. These two problems are unnecessary memory copies and interrupts [PIE01a]. The reasons for these two problems are as follows. A host system receives or transmits data as a set of packets. Excessive memory copying is a significant problem when the network speed approaches the speed of main memory. Moreover, each received packet needs to be filtered and demultiplexed to the correct application. This requires the moving of received packets from network interface card (NIC) to

applications. Therefore, avoiding memory copy is nearly impossible. Interrupts, on the other hand, are typically generated for each packet received or transmitted. For low-speed networks such as 10Mbps Ethernet this is not a significant problem, since the amount of interrupts is still only a few thousands per second even with small packets [MOG97]. The cost of handling interrupts at that rate was low enough and any normal system would spend only a fraction of its CPU time handling interrupts. On Gigabit Ethernet using the standard 1500 byte packets, an interrupt per packet would cause nearly 80000 interrupts per second [KIM01]. With smaller packets the problem is even worse.

In the following sections, we give a brief description about network interface model seen in most host systems. Then, we explain in detail the interrupt overhead and its related problems in interrupt-driven operating systems.

1.2.1 An Overview of Network Interface Model

The architecture of network interface system consists of several hardware and software interacting components in both the host computer and the NIC. Figure 1.2 depicts the major components seen in most Gigabit Ethernet network interface system.

We consider a typical host system where all the network interface functionality is performed by the operating system processes running in the kernel address space, while the application processes run in the user address space. We assume that the NIC is equipped with two DMA[1] engines. These engines are responsible for packets movement

[1] Direct Memory Access.

between NIC and host system memory. With Gigabit environment, the use of DMA becomes necessary in order to eliminate any CPU overhead involved in copying packets from (or to) NIC to (or from) host system memory. In this section we focus on the receive-part, where the interrupt overhead is more important.

Figure 1.2 shows the flow path of an incoming packet between the NIC, host memory, and applications. When a packet arrives at the NIC it gets temporarily stored in a local queue. Then, the NIC's device controller transfers the received packet to the host memory using Rx DMA engine[1]. After the incoming packet is placed into the host memory, the NIC generates a hardware interrupt to notify the OS of the arrival of a new packet. The OS invokes interrupt dispatcher to identify the nature of the interrupt and the corresponding device driver. The interrupt service routine (ISR), which is part of the network interface device driver, posts a software interrupt. Then, the software interrupt executes a filter function to enable posting the packet to the appropriate protocol processing routine (usually IP routine). As the protocol processing moves up the layers, the packet remains in the same kernel memory buffer that it was moved into, with only pointer manipulations between the protocol layers. Finally, the packet is moved from the kernel space to the user's address space[2], and then the recipient application is notified.

[1] The locations within host system memory reserved for received packets are indicated to Rx-DMA using Buffer Descriptor.
[2] This moving is performed within the context of the software interrupt.

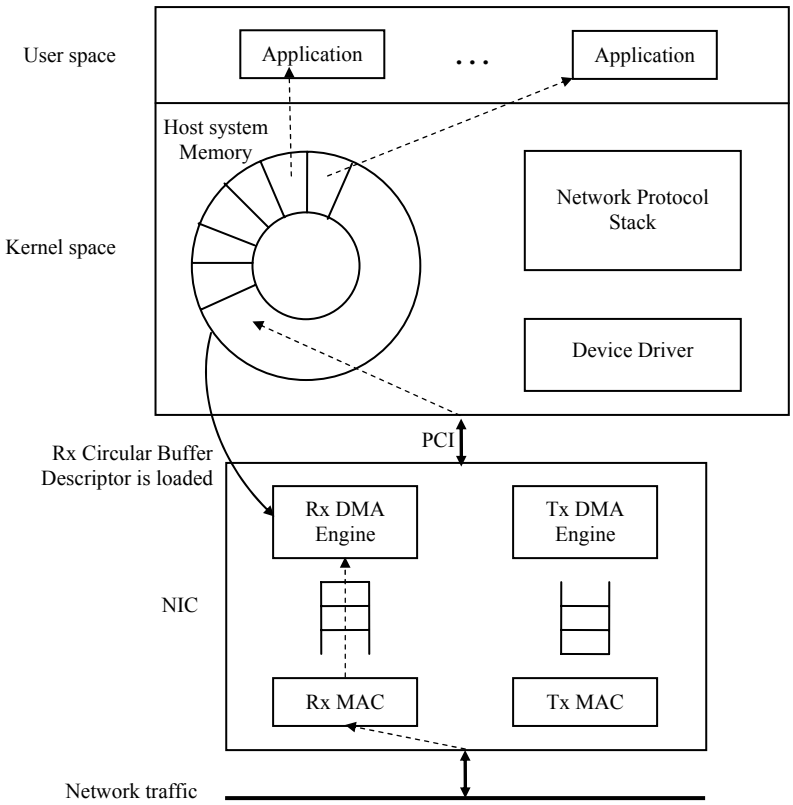


Figure 1.2: Typical network interface model for Gigabit Ethernet subsystem

1.2.2 Interrupt Handling Overhead

Most of the general-purpose operating systems utilize similar scheme for handling hardware interrupts. Generally, in most UNIX-based operating systems, each hardware interrupt requires the following steps [RUB01]:

1. Hardware and software context switching, preservation of CPU registers, and change of active processor stack.
2. Accessing the registers of hardware interrupt controller, and determination of the appropriate device driver interrupt service routine (ISR).

3. Updating the interrupt counters.
4. Processing the interrupt requests inside the designated ISR.
5. Upon completion of the ISR, system state is restored.

Execution time for steps 1, 2, 3, and 5 is mainly depending on CPU, memory, and system bus performance. In step 4, the execution time depends on the job of ISR. In old network interface system (no DMA support), the processing duration for step 4 is variable and it depends on the size of received packet. The main objective of ISR was moving the received packet from NIC buffer to the host memory. However, in our network interface system (with DMA support), the primary job of ISR is to notify the kernel of the arrival of a new packet. The notification only happens after the packet is successfully copied to the host system memory. Therefore, the ISR time (time spent to process all five steps) for one interrupt request is relatively constant for specific system hardware, and mostly unrelated to the network traffic or current system load.

As measured in [ARON00], a hardware interrupt with a null interrupt handler introduces an overhead of about 4 μs in a 500MHz Pentium III system running FreeBSD 2.2.6. On Gigabit Ethernet networking, the time between successive minimum sized packets (512-bytes) can be calculated as follows:

$$512\ bytes \times \frac{8\ bits}{1\ byte} \times \frac{1\ s}{1\times 10^{9}\ bits} \times \frac{1\ \mu s}{1\times 10^{-6}\ s} = 4.096 \mu \text{s} .$$

This means the CPU must handle an interrupt in less than 4 μs in order to keep the system responsive. However, the packet arrival rate can surpass the system packet processing rate which includes network protocol processing and interrupt handling. Therefore, interrupt overhead becomes an important overhead for interrupt-driven

systems that receive packets at gigabit speed from the NIC, and it is important to examine the possible schemes that can eliminate interrupt overhead.

1.2.3 Receive Livelock

In this section we describe briefly the phenomenon of receive livelock. Incoming network packets received at a host must either be forwarded to other hosts (as in the case of a router), or to application programs where they are consumed. The delivered system throughput is a measure of the rate at which such packets are processed successfully. Figure 1.3, adopted by [RAM93], shows the delivered system throughput as a function of offered input load. The figure illustrates that in the ideal case, no matter what the packet arrival rate, every incoming packet is processed. However, all practical systems have finite processing capacity, and cannot receive and process packets beyond a maximum rate. This rate is called the *Maximum Loss-Free Receive Rate* (MLFRR) [RAM93]. Such rate is an acceptable rate and is relatively flat after that. Under network input overload, a host can be swamped with receiving packets to the extent that the effective system throughput falls to zero. Such a situation, where a host has not crashed but is unable to perform useful work, such as delivering received packets to user processes or running other ready processes, is known as *receive livelock*. Similarly, under receive livelock, a router would be unable to forward packets to the outgoing interface, resulting in transmit starvation.

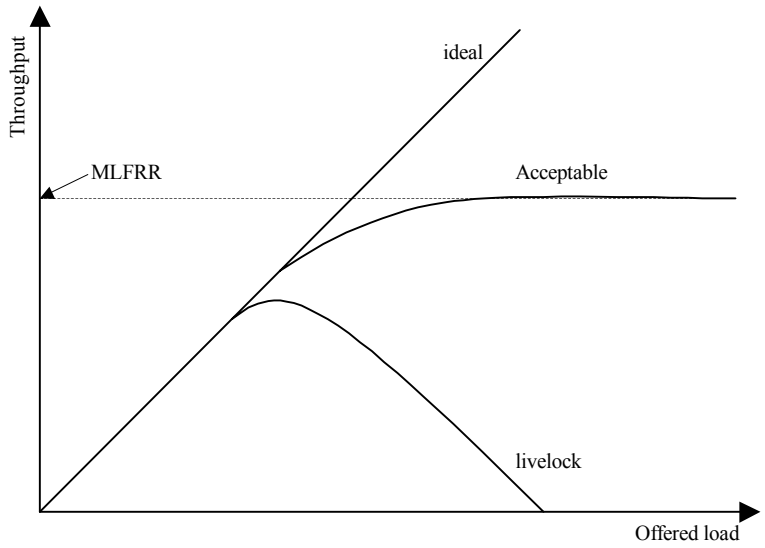


Figure 1.3: Possible behaviors of delivered throughput versus offered load

The main reason for receive livelock is that hardware interrupts (interrupts generated by NIC) are handled at a very high priority level, higher than software interrupts, or input threads that process the packet further up the protocol stack, or application processes. Generating interrupt upon packet arrival implies that the host must accept and process all incoming packets, regardless of whether the host system has sufficient processing capacity available to process them completely. As a consequence, under heavy network traffic, the system spends all of its resources handling interrupts. Since hardware interrupts and software interrupts have higher priority than application processes, the application queues will eventually fill because the receiving application no longer gets enough CPU time to consume the packets. At that point, packets are discarded when they reach application queue. As a result, starvation will occur for application processes.

As the load increases further, the software interrupts will eventually no longer keep up with the protocol processing, causing the IP queue to fill. The problem is that ISRs have strictly higher priority than software interrupts. Under overload, this will cause packets to be dropped from IP queue besides packet dropping in application queue.

In summary, interrupt-driven systems perform very badly under overload. High packet arrival rates can result in receive livelock, a situation where the host uses all of its capacity to receive incoming packets, and nothing else will be performed. In receive livelock, system throughput drops to zero, application processes and threads start to starve, and network latency increases rapidly.

1.3 Motivation

With emerging of Gigabit networks, achieving high performance communication becomes a challenge. Most modern operating systems depend on interrupts for event notifications. As noted earlier, interrupt-driven systems tend to perform very badly under Gigabit network environment.

Different solutions to eliminate interrupt overhead and resolve receive livelock problem have been proposed. Such solutions include interrupt coalescing, enabling and disabling interrupts, polling, jumbo frames, etc. The performance of these solutions has been studied experimentally. None of these solutions modeled and studied analytically the performance and behavior of system performance under heavy network loads.

1.4 Main Contributions

The main contributions of this thesis work are the followings:

- Conducting an extensive literature survey.
- Proposing analytical models to capture the impact of interrupt overhead on performance especially in systems with high arrival rates. These models can be utilized to understand and predict the performance of interrupt-driven systems and can be served as a reference model for comparing the performance of these proposed solutions to resolve the receive livelock condition. The models include Traditional scheme, Interrupt Coalescing scheme, and Enabling-Disabling Interrupt scheme.
- Proposing a novel metric to measure the overall system power.
- Simulation models for all interrupt handling schemes.
- Evaluation performance of interrupt handling schemes.
- Discussing some issues on design and implementation.

1.5 Organization of Thesis

This thesis is organized as follows. Chapter 2 gives an extensive literature survey of interrupt handling schemes. Chapter 3 presents analytical models to describe different optimization for interrupt handling. Chapter 4 presents simulation. Chapter 5 presents performance comparison between interrupt handling schemes and some implementation issues. Chapter 6 gives the conclusion and future work.

CHAPTER 2

LITERATURE REVIEW

In this chapter, we will discuss performance metrics used to evaluate interrupt handling schemes. Then, we will discuss different proposed solutions used to eliminate interrupt overhead and resolve receive livelock problem. We will also show how these solutions are implemented in host systems.

2.1 Performance Metrics

Before we introduce various schemes used for handling packet reception, we first have to define the following metrics, as they apply to the receive operation from the network interface system.

1. *Throughput*. We can define throughput as the rate at which packets successfully leave the network interface system (i.e. from the kernel buffer to the user space), or in other words, the rate at which the ultimate application can deliver packets from the network interface system. Therefore, any design of network interface system tries to maximize system throughput as much as possible.
2. *Latency*. Latency is the time duration between a packet arrival at the network interface system and its completion (i.e. its delivery to the ultimate

application). If the receive operation introduces more overhead, then the throughput of the system will decrease and the latency will increase. The latency can be larger than the overhead if the received packets are queued in the kernel buffer before they are delivered to the ultimate application. Thus, minimizing latency as much as possible is required.

3. *CPU Availability*. CPU availability is the percentage of time a server (or CPU) is available for user processes during a given interval of time. The design of the network interface influences the amount of CPU resources consumed for receiving data which we would like to minimize. When a host system is overloaded with incoming packets, it must continue to process other tasks, so that to allow applications to make use of the arriving packets. The operating system must fairly allocate CPU resources among packet reception and transmission, protocol stack processing, and application processing.
4. *Overall System Power*. The aforementioned goals; maximizing throughput, minimizing latency, and maximizing CPU availability are mutually contradictory in that all schemes to increase throughput result in decreased CPU availability with increased latency as well and vice versa. The advantage of the overall system power is that it gives the correct operating point that maximizes throughput, minimizes latency, and maximize CPU availability.

5. *Stability Condition*: Stability condition defines the maximum load after which the host system will not be stable due to buffer overflow, high traffic load, or due to interrupt overhead.
6. *Probability of loss*: The loss of packets from the host system memory is often the primary source of loss in local area networks. The probability of loss is impacted by the arrival rate of packets and the service rate. The probability of loss can give an indication of buffer availability and system load level.

It is worth noting that when we are going to design a system we have to specify the goal. The goal specifies which performance metric is more important than the others. For example, when we design a system to implement video streaming, then we focus on throughput more than other performance metrics.

2.2 Proposed Solutions to Reduce Interrupt Overhead

In this section, we present different proposed solutions for packet reception.

2.2.1 Interrupt Coalescing Scheme

Instead of generating an interrupt for each packet arrival, a group of packets will be notified to the operating system via a single interrupt request. This method is known as *interrupt coalescing* or *mitigation*.

Many modern NICs and device drivers adopt the idea of interrupt coalescing. Modern NICs configure interrupt coalescing through its registers. For example, TC9021

Ethernet NIC uses RxDMAIntCtrl register to configure interrupt coalescing. The interrupt frequency can be set based on either the number of packets received or after a fixed amount of time following the receipt of a packet (via another register field). Some NICs have intelligent hardware for packet reception. The NIC dynamically regulates its interrupt frequency based on traffic load. For example, when traffic is light, the NIC interrupts the host after receiving every packet to minimize packet delay. In heavy traffic, the NIC is able to optimize host efficiency by dynamically adjusting the CPU interrupt rate and issuing a single interrupt only when buffer space is low or its timer has expired.

Device drivers support interrupt coalescing through tuned parameters. For example, the Gigabit Ethernet driver on Linux that was developed for ACEnic NIC (produced by Alteon) uses two parameters; one parameter for transmission coalescing and the other for reception coalescing, to affect the times of interrupts from the NIC on the transmitting and receiving [CERN]. When Jumbo frames are enabled, the driver uses other two parameters to define interrupt coalescing on transmission and reception. Windows 2000 supports interrupt coalescing by specifying manually the number of interrupts per second. This can be found on the registry file under the following key:

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Tcpip \Parameters\Interfaces\<interface-name>\ MaxIRQperSec.

Another implementation of interrupt coalescing was proposed by [KIM01]. The solution is based on timer to generate interrupts. Therefore, the solution assumes that NIC has build-in timer chip. The solution has been implemented as follows. The device driver has been modified to operate in two modes: the interrupt mode and timer mode.

In interrupt mode, the module works in traditional manner as specified in GNU/Linux license. In timer mode, receive interrupts are totally disabled and NIC is equipped with a timer that generates an interrupt after passing a fix period of time. During time interval, the NIC may receive multiple packets after which they will be notified by a single interrupt. The NIC[1], on which this solution has been implemented, has timer-chip that clicks on a multiple of 81.92 μs.

When the driver module is loaded into kernel, the users can direct the driver to operate in the timer mode or in the interrupt mode. If the user selects the timer mode, he has the ability to configure the timer expiration period, i.e., the interval time between successive interrupts. This time must be a multiple of 81.92 μs. When the device driver started in timer mode, it resets NIC to only generate timer interrupts, transmission interrupts, and interrupts related to error reporting.

Indiresan et. al. [IND97] proposed another technique to implement dynamic interrupt coalescing called *Intelligent Interface Backoff*. The host system provides some feedback to the NIC of its current load, and the NIC determines its interrupt frequency based on this information. In light traffic, the host system behaves normally, i.e. it interrupts on every incoming packet. In heavy traffic, the NIC modulates its interrupt frequency on the basis of host's load. When the host indicates to the NIC that its load is increasing (and there is a backlog in the processing of incoming packets or executing other application), the NIC reduces its interrupt frequency. As the load on the host

[1] SMC Etherpower 10/100, based on DEC 2114 Tulip Ethernet controller chip.

decreases, the NIC increases its interrupt frequency until it reverts to the normal interrupt mode.

The host detects excess load or low load by using simple heuristic, which is buffer utilization. Since incoming packets are typically allocated a buffer, which is not freed until the packet is completely processed, high buffer utilization could indicate that packets are not being processed to completion fast enough. The host sends an overload indication when the buffer availability drops below 25%. The host sends this indication through device driver. Since the device driver typically issues commands to the NIC for every arriving packet, the NIC can adjust rapidly to overload situation. In addition, adding one field to these commands will not require much modification or increase the host-NIC interface overhead.

The solution has been implemented as follows. On initialization, the host issues a command to enable backoff feature on the NIC. This command has four parameters: *minimum backoff period* (I_{min}), *maximum backoff period* (I_{max}), *backoff factor* ($b > 1$), and *restore factor* ($r < 1$).

The interrupt frequency is determined by *backoff_period* value. Figure 2.1 depicts a pseudo-code describing how *backoff_period* value is updated.

Notice that when the backoff period falls below I_{min}, the NIC reverts to the normal interrupt mode and backoff period is bounded by I_{min} and I_{max} values. As backoff period increases the interrupt frequency decrease to adapt the host load. I_{min} represents the maximum interrupt frequency for the NIC, and sets an upper bound on the CPU capacity used for handling receive interrupts from that NIC. I_{max} represents the minimum interrupt frequency, and hence, the worst case latency for the host to start

processing packets on the interface. The backoff and restore factors bias an interface towards I_{max} and I_{min}, respectively. A large b makes the interface shed excess load rapidly, and a small r makes the interface reduce latency quickly as the offered load reduces.

```
set backoff_period to 0;
procedure update_backoff_period (indication as parameter)
begin
      if indication is overload then begin
            if backoff_period == 0 then
                  backoff_period = Imin
            else
                  if backoff_period + b < Imax then
                        backoff_period += b;
      end
      if indication is lower then begin
            if backoff_period <= Imin then
                  enable normal interrupt mode
                  return
            else
                  backoff_period -= r
      end
      wait for backoff_period  before interrupting again
end
```

Figure 2.1: Pseudo-code for Intelligence Backoff Interface

The performances of the above solutions have been studied experimentally. An experimental result from [HAS00] shows that small values of interrupt coalescing parameter give the best performance in terms of throughput. Interrupt coalescing

minimizes CPU utilization due to interrupt handling. However, interrupt coalescing increases the system response time.

2.2.2 Enabling-Disabling Interrupt Scheme

Another solution to eliminate interrupt overhead was proposed by Mogul and Ramakrishnan [MOG97]. The authors implemented a mechanism where interrupts are only used at low network load conditions, while in high loads the interrupts are disabled and a polling thread is scheduled for reading the network interface (or host system memory). Every time a poll is executed, a certain packet quota is specified, i.e. the maximum number of packets that can be read in that poll. The quota is used for fairness purposes when other tasks must also be permitted to make progress, so as to avoid livelock condition. If at the end of the polling some packets remain at the NIC, the polling thread is executed again after a few milliseconds. Otherwise, the system switches back to interrupts.

2.2.3 Polling Scheme

Rather than NIC controls receive operation, the host operating system periodically looks at NIC to see if it requires attention, and then invokes the handler accordingly. This method is known as *device status polling* [RIZZ02].

With polling, the asynchronous event notification concept based on hardware interrupt is completely abandonment, and OS initiates a read operation of a control NIC register after a predefine time duration. If one or more packets have arrived, then the

protocol stack routines are invoked to process the received packets. Since several packets may be read in the same poll and since the code to perform a poll is much shorter than the ISR, the receive overhead is reduced [DOV01].

One of the key advantages of polling is that it gives the OS a chance to control the amount of CPU spent in packet processing at protocol stack. This is done by adapting the maximum number of packets to be processed in each poll. The drawback of polling appears when the packet arrival rate is much lower than polling rate. In that case, packets are not guaranteed to be presented at each poll; the polls in which no packet is found in the system memory buffer (unsuccessful polls) increase the overall overhead of the network interface. Additionally, the latency of the receive operation increases because packets are queued in the system memory buffer until the polling event. Because of these two drawbacks, polling is not commonly used in general purpose systems. Polling is used however in systems that have a heavy network load, such as routers, bridges, firewalls, or file servers [DOV01].

There are two approaches to implement polling. One possible polling approach is to have a periodically scheduled kernel polling process [RIZZ02]. This approach, however, requires a context switch for each poll. But the overhead of this context switching is smaller than the cost of an interrupt. This solution can be implemented in multitasking operating systems.

The second approach is based on operating system soft clock [ARON00]. This clock causes a periodic interrupt that is used for time-slicing and other bookkeeping activities. Its period is the finest time slice and system clock granularity that the operating system allows. The most OS clock period was commonly set to 10

milliseconds [VAH96]. Although the polling period can be constrained to be a multiple of this period, the time granularity of adjusting the polling period would be too coarse, and the maximum latency that polling could introduce would be excessive (several tens of milliseconds) for many applications. More recently, some OS vendors have moved to a smaller clock interrupt period of 1 millisecond (e.g., Solaries 8).

Another solution for polling scheme was proposed by [MAQ96], namely, *Polling Watchdog*. Polling Watchdog is a hardware extension at the NIC that limits the generation of interrupts to the cases where explicit polling fails to handle the packets quickly. The basic idea is that when a packet arrives at the NIC, a timer starts counting. If the packet is not removed from the NIC through polling within a given amount of time (the watchdog timeout period T_{wdog}), the watchdog interrupts the CPU. T_{wdog} is set to around 50 μs, in order to strictly limit the maximum latency. In the EARTH-MANNA multiprocessor system on which this solution has been implemented, the cost of an interrupt is 4.5 μs, and the cost of a poll is 400 ns.

2.2.4 Interrupt-Polling Scheme

This scheme combines the advantages of interrupts and polling, i.e. it uses interrupts under low network load conditions and polling otherwise. Therefore, this scheme is expected to perform better than interrupts in terms of receive overhead due to interrupt handling and better than polling in terms of receive latency due to unnecessary packet queuing.

One implementation for this scheme was performed for Windows NT platform [HAN97]. The implementation simply turns off interrupts and uses polling under high traffic load. When the traffic load decreases, it turns the interrupts back. The traffic load is captured by user thread starvation. The authors list 3 possibilities to detect user thread starvation: length of network data queue, interrupt rate, and amount of time spent processing interrupts. The authors implement the interrupt processing time for indication of system load. The measurement as obtained experimentally using some network tools.

Another implementation that combines interrupts with polling, namely, *Hybrid Interrupt-Polling* (*HIP*). The basic idea of HIP is to adaptively switch between the use of interrupts and polling based on the observed rate of packet arrivals. Specifically, if the packet arrivals are frequent and predictable, the receive mechanism operates in polling mode and interrupts are disabled. In this mode the polling period is set based on the predicted packet interarrival time. However, to bound the receive latency, the polling period is not allowed to exceed a pre-determined limit. On other hand, if the packet arrivals are infrequent, less predictable, or if the number of consecutive unsuccessful polls exceeds a threshold, the receive mechanism operates in the interrupt mode. In this mode, the polling operating is stopped and the interrupts are enabled.

2.2.5 Jumbo Frames

Jumbo frames are frames that are longer than the standard Ethernet (IEEE 802.3) frame length of 1,518 bytes. The frame size definition for jumbo frames is vendor-

specific because Jumbo frames are not part of the IEEE standard. The most commonly used Jumbo frame sizes are 9,018 bytes and higher. Jumbo frames are not a CSMA/CD modification; in fact, they only work in a full duplex environment.

Jumbo frames maintain the same media access control (MAC), frame structure, and frame check sequencing mechanism used for traditional Ethernet frames. Only the payload portion of the frame is extended.

The choice of 9000 bytes for the Jumbo frames payload length is to provide a good compromise between frame efficiency, frame check sequence effectiveness, and host protocol stack efficiency. Most IP protocol stacks can be configured to support maximum transmission units (MTUs) of up to 64 Kbytes. But Ethernet error detection techniques provide a practical upper limit on frame size. Due to the nature of the CRC-32 algorithm, the probability of an undetected error is essentially unchanged until frames exceed approximately 12,000 bytes. Thus, to maintain the same undetected bit error rate (BER) as standard Ethernet, Jumbo frame sizes should not exceed 12,000 bytes. On the other hand, the maximum size for a network file system (NFS) datagram is typically around 8 Kbytes. To ensure that an entire NFS datagram can be transmitted in one frame, jumbo frames should be at least 8 Kbytes. Moreover, host protocol stacks operate most efficiently when working with data that is an integer multiple of the page size of the operating system. For most operating systems this is 4096 (4K) bytes. Using a 9000-byte as frame size allows the carriage of 2 pages of user data (8192 bytes) plus the various transport, network and data link headers.

CHAPTER 3

MODELING AND ANALYSIS

In this chapter, we provide an analytical study of packet reception through network card (NIC) based on queuing theory. An analytic model is one that can be expressed as a set of equations which can be solved in order to measure OS performance. For many practical real-world problems, analytic models based on queuing theory provide a reasonable approximation to real system.

The objective of our analysis is to study the impact of interrupt overhead on system performance in terms of system throughput, system latency, CPU availability, and overall system power. We first model an ideal situation where the interrupt overhead is ignored to determine the optimal system performance.

Next, we model an interrupt-driven system in which the interrupt overhead is taken into account. Receive livelock phenomenon can be analyzed and determined.

Finally, we study analytically the system performance of the proposed solutions for resolving and eliminating the receive livelock problem. These solutions are Interrupt Coalescing, Enabling-Disabling Interrupt, and Polling.

3.1 Queuing Theory

Queuing analysis is one of the most important tools for those involved with computer and network analysis. Queuing theory provides the basic tools for modeling and analyzing the system. By using queuing analysis, one can study and evaluate the system performance in terms of some parameters such as average number of packets in the system, system throughput, mean response time, and so on.

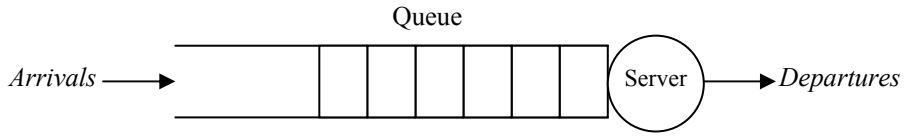


Figure 3.1: Queuing model for the system

Figure 3.1 illustrates a queuing model for the system. Packets are randomly arriving to the system from the NIC. The queue represents the host system memory in which all arrival packets stored in this queue. The server represents the CPU that is responsible to process all received packets. Packets are processed either in ISR or in protocol stack. Packets are served by first-come-first-serve order.

Formally, queuing systems are characterized by stochastic characteristics. These characteristics are the arrival process, the service time of the server, the number of servers, the system capacity, as well as some special properties of the system. These stochastic characteristics can be summarized by using Kendall notation:

$$A / B / s / k$$

where *A* refers to the distribution of the time between two successive arrivals, *B* refers to the distribution of service time, *s* refers to the number of servers, and *k* is an upper bound on the number of packets in the system.

3.1.1 Notations and Assumptions

Let us consider an arrival process $\{N(t), t \geq 0\}$, where $N(t)$ denotes the number of packets in the system up to time t with $N(0) = 0$, as *stochastic process* which varies in time. We wish to predict its future behavior with the aid of a certain amount of probability. This probability is governed by a random distribution or a set of random distributions. For our system, there are three random distributions that determine the behavior of our stochastic process: the time between successive arrivals, packet length, service time for protocol processing and ISR time.

Let λ be the average incoming packet arrival rate. Therefore, $1/\lambda$ is the time between successive arrivals (interarrival time). Similarly, let μ be the average protocol processing rate by the kernel. Therefore, $1/\mu$ is the time it takes the system to process the incoming packet and deliver it to an application program. This time includes primarily the network protocol stack processing by the kernel, excluding any interrupt handling. However, the interrupt handling time will be denoted as T_{ISR}, which is basically the interrupt service routine time for handling incoming packet. The average interrupt service routine rate is denoted as *r*. We will also denote ρ as a measure of the traffic intensity or system load and is denoted as λ / μ.

An important class of stochastic processes is *Markov processes*. This class of processes has some special properties that make them manageable to treat mathematically. A Markov process is a random processes where the value of the random variable $N(t)$ at time t_n depends only on its immediate past value at time t_{n-1}. Markov processes assume that interarrival times and service times obey the exponential distribution or, equivalently, that the arrival rate and service rate follow a Poisson distribution [GRO98].

In a Markov process, $N(t)$ represents the *state* of the system at a given time t. If the sample space, $N(t)$, is discrete, then Markov process is called *Markov chain*. Markov chains can be visualized by drawing *rate-transition diagram* that displays the rate flow between different states in the Markov chain. Then, the Markov chain can be summarized in one matrix called *intensity matrix* and it is denoted by **Q**.

Now, given

$$\mathbf{Q} = \begin{pmatrix} -q_0 & q_{01} & q_{02} & \cdots \\ q_{10} & -q_1 & q_{12} & \cdots \\ q_{20} & q_{21} & -q_2 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

where q_{ij}, for $i \neq j$, is the intensity transition from state i to j and $q_i = \sum_{i \neq j} q_{ij}$. We wish to find the steady-state probabilities, p_i, of the Markov chain where p_i is the probability that the system will be at state i. Let us represent the steady-state probabilities as a vector $\boldsymbol{p}$, then the equation:

$$\mathbf{0} = \boldsymbol{p}\mathbf{Q} \qquad (3\text{-}1)$$

constructs well-known equations which are called the *stationary equations* of the Markov chain. Together with the boundary condition that $\sum_i p_i = 1$, we can obtain p_i with some special mathematical transformation.

Knowing these probabilities, we can obtain a lot of information about system behavior. For example, p_0 is the probability that the system at state zero, or equivalently, the probability that the system is idle. Therefore, $1 - p_0$ represents the probability that the system is busy.

In order to simplify the analysis, we will assume that all packets arrive to the NIC have fixed size length. Moreover, we also assume that we have only one CPU (server) in the system. When the system has a single-server with Poisson arrivals and exponential service times, then our system can be modeled as M/M/1.

Our analytical models follow the architecture of network interface mentioned in section 1.2.1. The Rx-DMA is responsible to move received packets from NIC buffer to the host system memory without the intervention of CPU.

Finally, we assume that the kernel protocol processing for packets will continue as long as there are packets available in the host system memory. This means packets could be processed in kernel by protocol stack routines without interrupt notification. In this situation, we say that the system is running at *full speed*.

3.1.2 Performance Metrics

1. **Throughput**. Throughput (γ) can be calculated by using the following general equation [TRI98]:

$$\gamma = \sum_i \mu_i p_i \,. \qquad (3\text{-}2)$$

If $\mu_i = \mu$ for all $i > 0$, then

$$\gamma = \sum_i \mu_i p_i = \mu \sum_i p_i = \mu(1 - p_0)\,. \qquad (3\text{-}3)$$

2. **Latency**. Latency (R) is the mean response time of the system. It can be calculated by using Little's theorem [KLIE71]:

$$R = \frac{E(n)}{\lambda}\,. \qquad (3\text{-}4)$$

where $E(n)$ is the expected number of packets in the system where its value can be calculated by using the following general equation:

$$E(n) = \sum_n n\, p_n \,. \qquad (3\text{-}5)$$

3. **CPU availability**. CPU availability (V) for user processes measures the fraction of time that CPU is available for other processes. The probability that the system is in state 0 p_0 represents a better metric for CPU availability.

4. **Overall system power**. We propose a novel single performance system metric to measure and evaluate the overall system performance. The overall system power (P)

is a single metric that integrates a number of performance metrics. The integrated metrics include the above three parameters which are system throughput, system latency, and CPU availability. This metric is similar to [GIES78], however, they consider only two parameters to determine the network power; throughput and latency, since these two parameters are quite enough to measure the network performance. In our system, we have to consider a third parameter which is starvation of lower priority processes or CPU availability to process these processes as we have mentioned previously. System throughput and CPU availability give more power to the design of network interface while system latency reduces overall power. Therefore, our proposed metric will be expressed as

$$P = \frac{\gamma^a V^b}{R^c}. \qquad (3\text{-}6)$$

where, a, b, and c are tunable parameters. Notice that the overall power (P) will increase when the system throughput and CPU availability are increased, and system latency is decreased. Normally, a, b, and c are equal to 1 which gives equal weight to all three parameters.

A particular point of interest is finding the maximum power point. This point is also the optimal operating point which gives maximum throughput, maximum CPU availability, and minimum system latency. The maximum power point is defined as the "knee" point for overall system power [JAIN88]. The peak of the overall power curve occurs at the knee point. Therefore, to obtain this point, we take the derivative of the

power function with respect to λ, and solving the derivative after making it equals to zero.

5. **Stability condition**. Another particular point of interest is finding the stability condition of the system. The stability condition is the situation where $\rho < 1$ or is defined as the "cliff" point for the system throughput [JAIN88]. It is where the throughput starts falling to zero as the system load increases.

6. **Loss probability**. In any finite buffer system, the loss probability (P_L) is a measure of the number of packets being lost, or in other words, it is the probability that the buffer is full at an arbitrary point in time. This means that if the system memory is of size B, the loss probability is given by p_B. Loss probability is important because it can be used to determine the proper memory buffer size that must be allocated for a given system in order to reduce the packet loss.

3.2 Ideal System Model

This section presents analysis for the ideal situation in which the overhead involved in generating interrupts is totally ignored. We can simply model such a system as an M/M/1/B queuing model with a Poisson packet arrival rate λ and an exponential protocol processing service time $1/\mu$. Note that in this case the system packet processing time is equal to protocol stack processing time since T_{ISR} is equal to zero. B is the maximum size the system memory buffer can hold. M/M/1/B queuing model is chosen as opposed to M/M/1 since we can have arrival rate go beyond the service rate. This

assumption is true in Gigabit environment where, under heavy load, λ can be very high compared to μ.

In M/M/1/B model, the equation for p_n is given by

$$p_n = \begin{cases} \dfrac{(1-\rho)\rho^n}{1-\rho^{B+1}}, & (\rho \neq 1) \\ \dfrac{1}{B+1}. & (\rho = 1) \end{cases} \qquad 0 \le n \le B \qquad (3\text{-}7)$$

Therefore, knowing these probabilities, we can now examine the system behavior for the ideal situation.

3.2.1 Performance Metrics

1. System throughput. By using Equation (3-2), the Ideal system throughput can be expressed as

$$\gamma = \mu \sum_{n=1}^{B} p_n = \mu (1 - p_0). \qquad (3\text{-}8)$$

where p_0 can be calculated by direct substitution to Equation (3-7).

2. System latency. To obtain the mean response time or Ideal system latency, we have to obtain the average number of packets in the system which is given by

$$E(n) = \frac{\rho}{1-\rho} - \frac{B+1}{1-\rho^{B+1}} \rho^{B+1}. \qquad (3\text{-}9)$$

Therefore, system latency is

$$R = \frac{E(n)}{\lambda_{eff}} = \frac{E(n)}{\lambda(1-p_B)}, \qquad (3\text{-}10)$$

where λ_{eff} is the effective arrival rate which is the average rate of packets actually entering the system, and p_B is the probability of loss.

3. CPU availability. CPU availability can be expressed as

$$V = \begin{cases} \dfrac{1-\rho}{1-\rho^{B+1}} & (\rho \neq 1), \\ \dfrac{1}{B+1} & (\rho = 1) \end{cases}. \qquad (3\text{-}11)$$

4. Overall system power. The system is stable whenever $\rho < 1$. Hence, it is suitable to model our system as M/M/1 in order to express the function of overall system power. For this case, the throughput, CPU availability, and latency are expressed as

$$\gamma(\lambda) = \mu(1-p_0) = \mu\left(1-\left(1-\frac{\lambda}{\mu}\right)\right) = \mu\left(\frac{\lambda}{\mu}\right) = \lambda,$$

$$V(\lambda) = 1-\rho = \frac{\mu-\lambda}{\mu},$$

and

$$R(\lambda) = \frac{E(n)}{\lambda} = \frac{\frac{\lambda}{\mu-\lambda}}{\lambda} = \frac{1}{\mu-\lambda}.$$

Therefore,

$$P(\lambda) = \frac{\gamma(\lambda)^a V(\lambda)^b}{R(\lambda)^c} = \frac{\lambda^a}{\mu^b}(\mu-\lambda)^{b+c}. \qquad (3\text{-}12)$$

Taking the derivative of $P(\lambda)$,

$$\frac{dP(\lambda)}{d\lambda} = a\frac{\lambda^{a-1}}{\mu^b}(\mu-\lambda)^{b+c} - (b+c)\frac{\lambda^a}{\mu^b}(\mu-\lambda)^{b+c-1}.$$

Putting $dP/d\lambda = 0$, we have

$$\frac{\lambda^{a-1}}{\mu^b}(\mu-\lambda)^{b+c-1}\left[a(\mu-\lambda)-(b+c)\lambda\right] = 0.$$

The above equation has three solutions. Two of them, which are $\lambda = 0$ and $\lambda = \mu$, are rejected because these points represent the least power values. The third solution which is $a(\mu-\lambda)-(b+c)\lambda = 0$ represents the maximum power point. Thus, the optimal operating point for Ideal model occurs at

$$\lambda = \left(\frac{a}{a+b+c}\right)\mu. \qquad (3\text{-}13)$$

Notice that, if a, b, and c are equal to 1, then the optimal operating point occurs at $\rho = 1/3$.

5. Stability condition: The system will be stable whenever $\lambda < \mu$.

3.2.2 Numerical Results

In this section, we give some numerical examples to study the behavior of the Ideal system. The system performance is studied as a function of traffic intensity ρ. For all of these results, we fix μ to 1 and B to size a size of 100.

Figure 3.2 depicts the graph of Ideal system throughput. We see that the system throughput increases as the arrival rate increases up to a point after which the system throughput remains constant because the server is processing packets at its maximum

capacity. Figure 3.3 depicts the graph of Ideal system latency. At $\rho = 1$, the server becomes saturated, working 100% of its time and the latency becomes infinity. Notice that, as shown in the figure, latency increased rapidly near system saturation. Figure 3.4 depicts the CPU availability for lower priority processes. Notice that, the CPU utilization due to packet processing increases as traffic intensity increases. When ρ is greater than one, the server works at full speed and consumes all CPU time. Hence, CPU availability for other processes will be diminished and lower priority processes start starving.

Figure 3.5 depicts the overall power of the Ideal system where all tunable parameters are equal to 1. As shown, at the beginning the system power increases as system load increases. Then, the overall power reaches its maximum value in which the system gives the optimal result. After this point, the power of the system starts decreasing until it reaches zero. Figure 3.5 shows that the maximum system power is when $\rho = 0.33$. This point matches exactly the point derived by equation (3-13) for finding λ that gives the maximum power point if we substitute *a*, *b*, and *c* by 1.

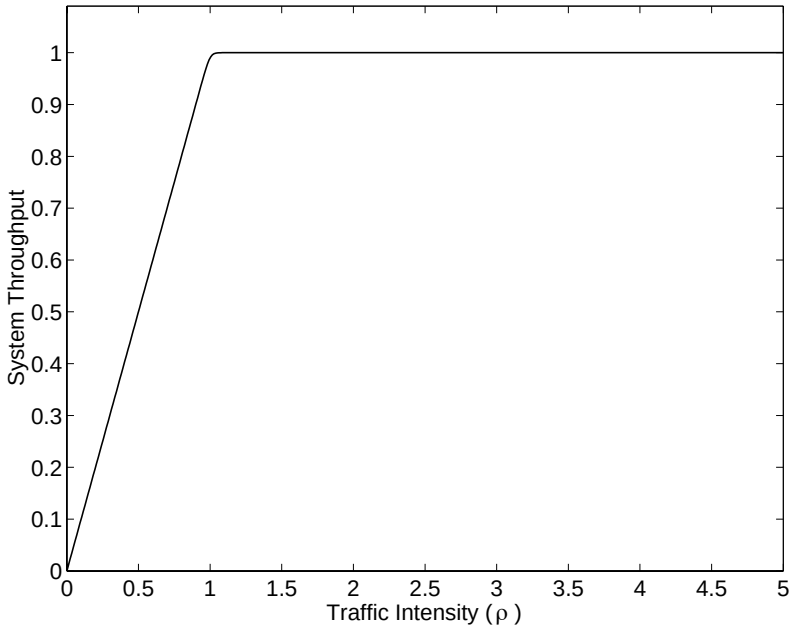


Figure 3.2: System throughput for Ideal system

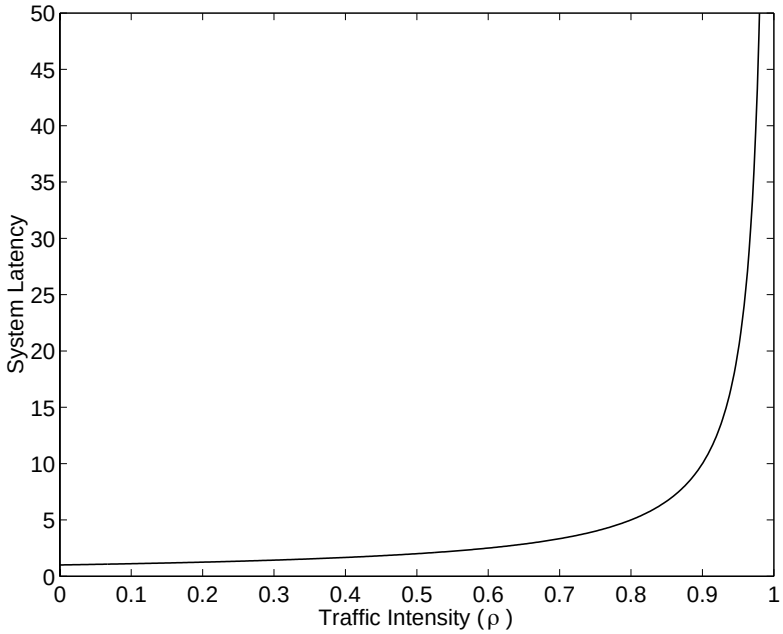


Figure 3.3: System latency for Ideal system

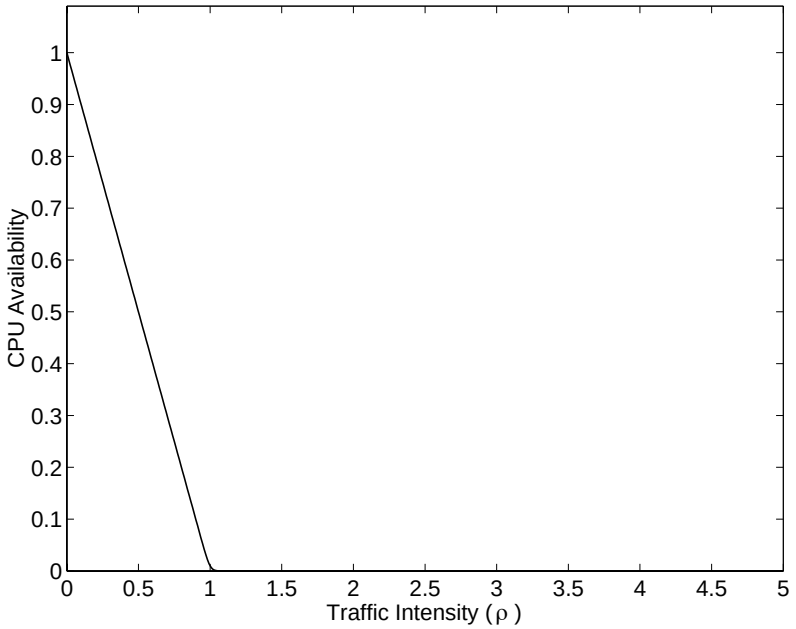


Figure 3.4: CPU availability of user processes for Ideal system

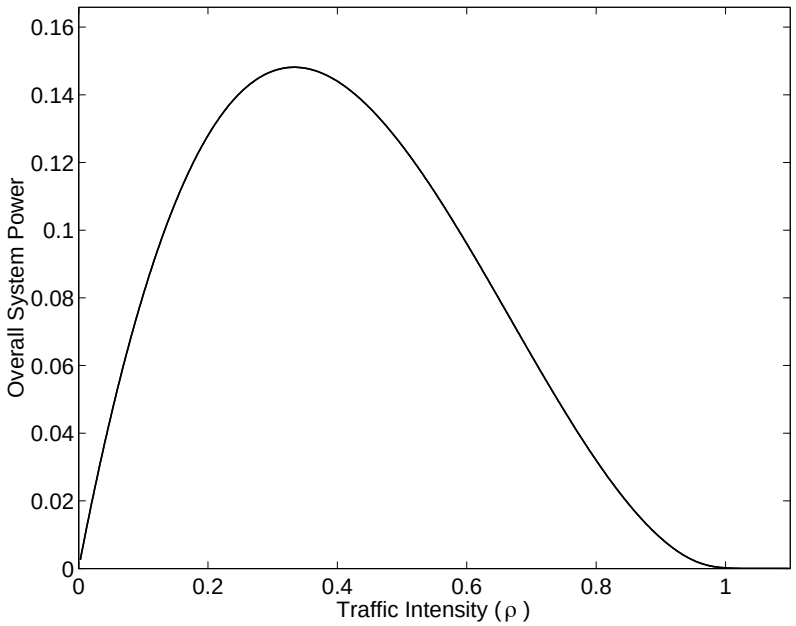


Figure 3.5: Overall system power for Ideal system

3.3 Interrupt-Driven System Models

Modeling an interrupt-driven system is a challenging task especially when we consider the Gigabit networking environment with $\rho > 1$. The most critical task in modeling an interrupt-driven system is determining the actual service time which we call it the *effective service time*. The effective service time is the total time to process a packet up to completion, inclusive of ISR disruption.

One can simply say that the effective service time is the time duration to process incoming packet in the kernel protocol stack and delivering it to the ultimate application plus the time duration to execute an ISR. However, this is not always true. We have two situations where the effective service time will be effected:

- If a new packet arrives while servicing a packet in the kernel protocol stack, then the effective service time will be increased by T_{ISR}. This is true, since ISR preempts any processes in the kernel.
- With Gigabit environment, a packet or multiple packets may arrive during execution of an ISR. In this case, we will have batched or *masked-off* interrupts and the packets will be queued into the system with effectively one T_{ISR} disrupting the service time.

In order to determine the effective service time, we use for our analysis a deterministic model where interarrival time, service time for protocol processing, and ISR time are all fixed.

3.3.1 Deterministic Model

First, we start by considering the case where $1/\lambda > T_{ISR}$, i.e., each incoming packet will generate an interrupt as illustrated in Figure 3.6.

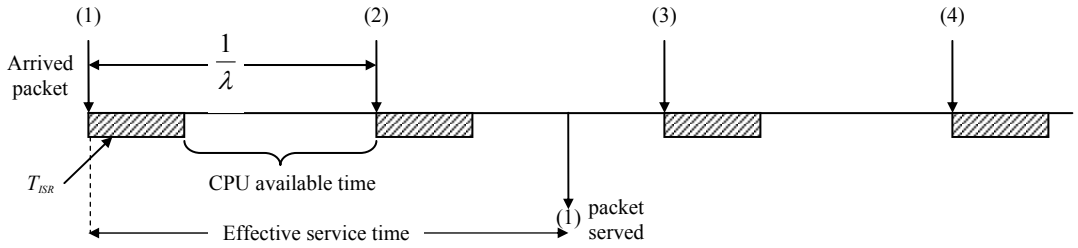


Figure 3.6: Timeline for a deterministic model where $1/\lambda > T_{ISR}$

One simply can calculate the effective service time for packet processing as $1/\mu + nT_{ISR}$, where $n = 0,1,2,\cdots$. For example, the effective service time for the first packet in Figure 3.6 is $1/\mu + 2T_{ISR}$. However, this way can be more complicated to obtain the value of *n* especially when we have masked-off interrupts. Rather we use the CPU availability *exclusive* of any ISR disruption. This means the available CPU time to process a packet in protocol stack is the time duration between successive ISRs. Mathematically, expressing the available CPU time for packet processing is straightforward. Notice that, for each $1/\lambda$ time unit, the available CPU time is $1/\lambda - T_{ISR}$ time unit. Therefore, for each 1 time unit, the available CPU time is

$$\text{Available CPU Time} = \frac{1/\lambda - T_{ISR}}{1/\lambda} = 1 - \lambda T_{ISR}. \qquad (3\text{-}14)$$

Equation (3-14) represents the percentage of CPU availability excluding any ISR disruption for a given arrival rate λ. Now, this equation can be used to calculate the service time to process a single packet. This time is what we call it the *effective service time for packet processing* and we will denote it by $1/\mu'$. If the service time for protocol processing is $1/\mu$, then the effective service time for packet processing is

$$1/\mu' = \frac{1/\mu}{1-\lambda T_{ISR}}. \qquad (3\text{-}15)$$

Hence, the effective service rate for packet processing is

$$\mu' = \mu(1-\lambda T_{ISR}). \qquad (3\text{-}16)$$

Notice that the effective service rate for packet processing is the mean service rate for protocol processing multiplied by the percentage of CPU availability for protocol processing.

Next, we consider the case where $1/\lambda < T_{ISR}$ and two packets arrive within the same interrupt as illustrated in Figure 3.7

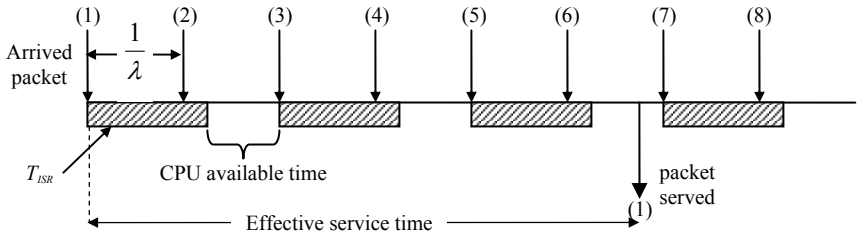


Figure 3.7: Timeline for a deterministic model where $1/\lambda < T_{ISR}$

Now, for each $2/\lambda$ time unit, the CPU available time is $2/\lambda - T_{ISR}$ time unit. Therefore, for 1 time unit, the percentage of CPU availability for packet processing is

$$\%\,\text{CPU Available Time} = \frac{2/\lambda - T_{ISR}}{2/\lambda} = 1 - \frac{\lambda}{2} T_{ISR}\,. \qquad (3\text{-}17)$$

Hence, the effective service rate for packet processing is

$$\mu' = \mu(1 - \frac{\lambda}{2} T_{ISR})\,. \qquad (3\text{-}18)$$

We give a detailed explanation about the relation between packet arrival rate and CPU availability. This relation is illustrated in Figure 3.8.

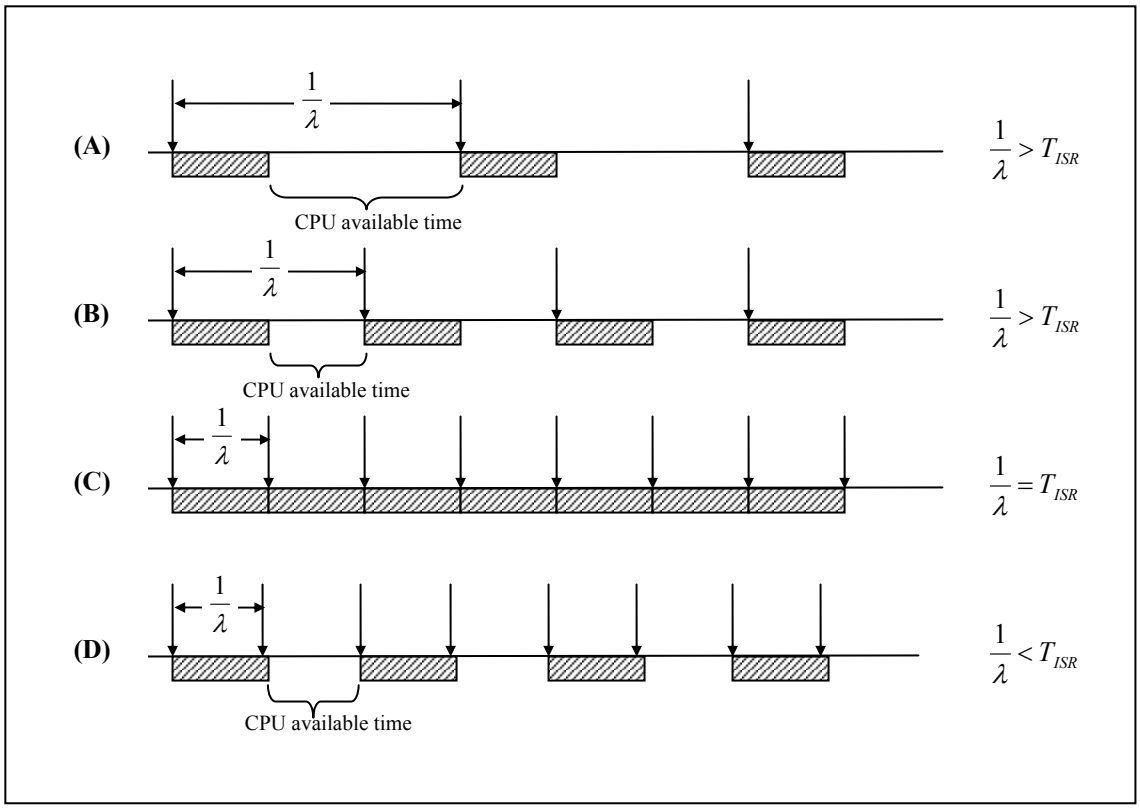


Figure 3.8: Timelines shows different amount of CPU available times

Figure 3.8 shows four timelines with different interarrival times. When the arrival rate is too low, as shown in Figure 3.8 (A), the available time is big enough to allow processing a packet up to completion. As the arrival rate increases, Figure 3.8 (B), the CPU available time decreases and; consequently, effective service time is increased in order to complete processing a packet. When a new packet arrives immediately after returning form interrupt handler, a new interrupt handler will be executed and the CPU available time becomes almost zero as shown in Figure 3.8 (C). Now, if the average arrival rate increases slightly such that the next coming packet arrives while the system execution is about to finish the current ISR (the ISR of the previous packet). In that case, as shown in Figure 3.8 (D), the second packet will be received without generating an interrupt. This means we have batch interrupts and the system restarts having some available time until the arrival of a new packet. Obviously, the average amount of available time will not exceed T_{ISR}.

3.3.1.1 General Formula for Effective Service Rate

Notice that, in Equation (3-18), the value of 2 represents the number of packets arrived within one ISR. Generally, this number can be expressed as $\lceil \lambda T_{ISR} \rceil$ where $\lceil \ \rceil$ denotes the ceiling number of λT_{ISR}. Thus, a general formula for the percentage of CPU availability can be expressed as

$$\%\text{CPU Available Time} = 1 - \frac{\lambda}{\lceil \lambda T_{ISR} \rceil} T_{ISR}. \qquad (3\text{-}19)$$

And, the effective service rate for packet processing can be expressed as

$$\mu' = \mu(1 - \frac{\lambda}{\lceil \lambda T_{ISR} \rceil} T_{ISR}) . \qquad (3\text{-}20)$$

Figure 3.9 depicts the relation between packet arrival rate and the CPU available time. The graph has been plotted where T_{ISR} = 0.3. As clearly shown, as packet arrival rate increases the CPU available time decreases until the CPU available time becomes zero. This is similar to Figure 3.8 (C). Obviously, this point is $1/\lambda = T_{ISR}$. After this point, batch arrival of size two will occur and, consequently, the interrupt overhead will be reduced and the CPU available time jumps up and the system can perform a useful work to process the incoming packet. As packet arrival rate keeps increasing, the CPU available time continues to decrease until it becomes zero again, in that case, when $2/\lambda = T_{ISR}$. If packet arrival rate increases slightly, a batch arrival of size three will be notified with only one ISR and CPU available time jumps up and the previous scenario will be repeated.

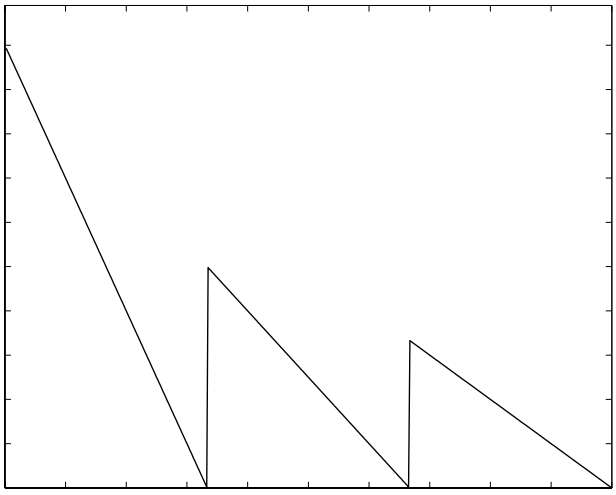


Figure 3.9: CPU available time for protocol processing versus packet arrival rate for D/D/1

The system throughput of the D/D/1 can be expressed as follows:

$$\gamma = \begin{cases} \lambda & \lambda < \mu' \\ \mu' & \lambda > \mu' \end{cases} \quad (3\text{-}21)$$

If we consider T_{ISR} = 0.3 unit of time, then the throughput of the system is shown in Figure 3.10. System throughput starts increasing as traffic intensity increases because the effective available time is enough to process the incoming packet up to completion. The system throughput keeps increasing until the CPU available time becomes almost equal to the system service time after which the throughput starts decreasing. The system throughput keeps decreasing since the amount of CPU available time keeps decreasing until no more available time to process incoming packets. Therefore, the system throughput becomes zero (as shown in Figure 3.8 (C)). At this state, the system will receive livelock.

If we extend packet arrival rate, we will see the following behavior as shown in Figure 3.11. We notice that the system throughput will not stay at zero as traffic intensity increases, instead, the system throughput will start oscillating above zero before it finally resets to zero as ρ grows larger and larger.

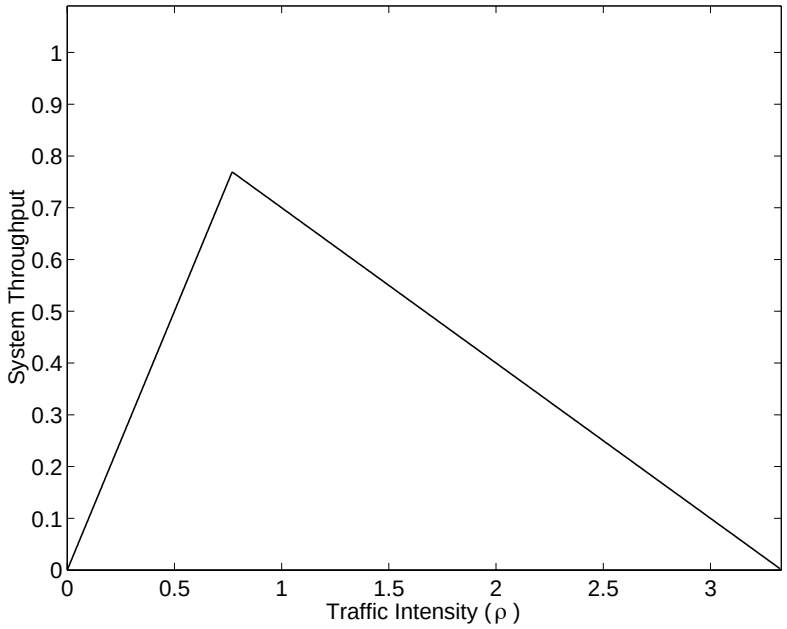


Figure 3.10: System throughput in Deterministic model

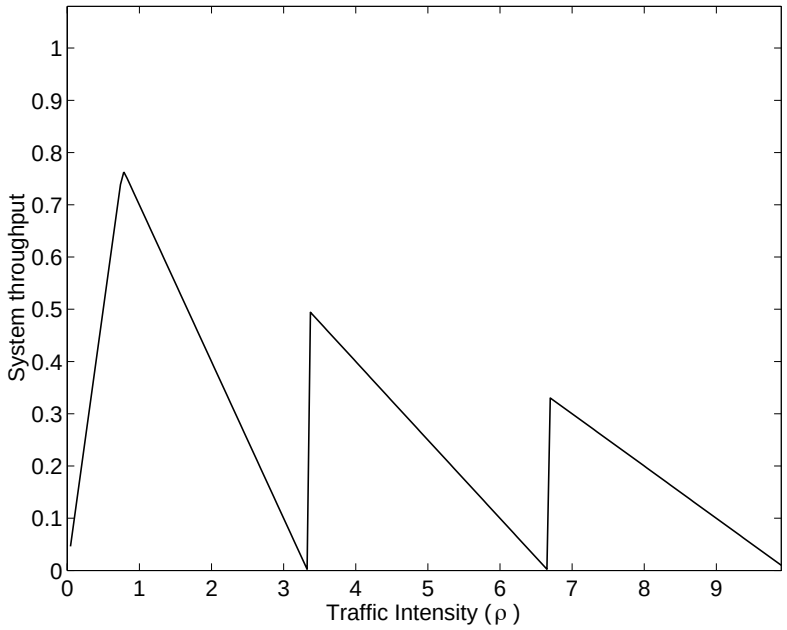


Figure 3.11: Effect of large values of ρ on system throughput in Deterministic model

3.3.2 Markovian Modeling

This section presents two different analytical techniques for studying system behavior of a traditional interrupt-driven kernel. The first technique exploits the idea of determining effective service rate through the percentage of CPU availability for protocol processing. The second technique uses pure Markovian process.

One may think that such an interrupt-driven system can be simply modeled as priority queuing system with preemption in which there are two arrivals of different priorities. The first arrival constitutes that for ISRs and has the higher priority. The second arrival is the arrival for incoming packets, and has the lower priority. However this is an invalid model because, as we mentioned before, ISR servicing is not counted for every packet arrival. The ISR servicing is ignored if the system is servicing another interrupt of the same level. In other words, if the system is currently executing another ISR, the new ISR which is of the same priority interrupt level will be masked off and there will be no service for it.

3.3.2.1 First Technique: Effective Service Time

In this section, we find the mean effective service time for processing packets in the kernel protocol stack. We first find the formula for the mean effective service time. Knowing this formula, the system can be modeled as an M/G/1 queue with a Poisson packet arrival rate of λ and a mean effective service rate of μ' that takes a general distribution.

One can express the mean effective service rate as

$$\mu' = \mu \times (\%\ \text{CPU Availability for protocol processing}). \quad (3\text{-}22)$$

In order to determine the CPU availability percentage for protocol processing, we use a Markov chain to model the CPU usage for ISR handling, as illustrated in Figure 3.12. We assume that T_{ISR} is exponentially distributed with mean $T_{ISR} = 1/r$. The process space has state $(0,0)$ and states $(1,n)$. State $(0,0)$ represents the state where the CPU is available for protocol processing. State $(1,n)$ with $n \geq 0$ represents the state where the CPU is busy handling interrupts. n denotes the number of packet arrivals that are being batched or masked off during T_{ISR}. Note that when process in state $(1,0)$, this means there are no interrupts being masked off and the CPU is handling a single interrupt.

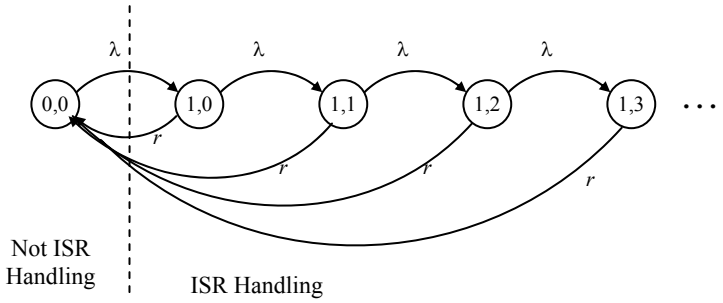


Figure 3.12: Rate-transition diagram to model CPU usage for Traditional scheme

The steady-state difference equations can be derived form Equation (3-1) where $\boldsymbol{p} = \{p_{0,0}, p_{1,0}, p_{1,1}, p_{1,2}, \cdots\}$ and $\boldsymbol{Q}$ is defined as follows:

$$\boldsymbol{Q}=\begin{bmatrix} -\lambda & \lambda & 0 & 0 & 0 & \cdots \\ r & -(\lambda+r) & \lambda & 0 & 0 & \cdots \\ r & 0 & -(\lambda+r) & \lambda & 0 & \cdots \\ r & 0 & 0 & -(\lambda+r) & \lambda & \cdots \\ r & 0 & 0 & 0 & -(\lambda+r) & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \end{bmatrix}$$

This will yield to

$$-\lambda p_{0,0} + r\,p_{1,0} + r\,p_{1,1} + r\,p_{1,2} + \cdots = 0\,. \qquad (3\text{-}23)$$

Since we have $p_{0,0} + \sum_{i=0}^{\infty} p_{1,i} = 1$, then Equation (3-23) can be rewritten as follows:

$$-\lambda p_{0,0} + r\,(p_{1,0} + p_{1,1} + p_{1,2} + \cdots) = 0\,,$$

$$-\lambda p_{0,0} + r\,(1 - p_{0,0}) = 0 \quad \Rightarrow \quad -\lambda p_{0,0} + r\,p_{0,0} + r = 0\,.$$

Solving for $p_{0,0}$, we thus have

$$p_0 = \frac{r}{\lambda + r}, \qquad (3\text{-}24)$$

and

$$1 - p_0 = \frac{\lambda}{\lambda + r}\,. \qquad (3\text{-}25)$$

Therefore, CPU cost of ISR handling is $\lambda/(\lambda+r)$ where as CPU availability for other processes including protocol stack processes is $r/(\lambda+r)$. Notice that the percentage of CPU availability is decreased as packet arrival is increased. The amount

of CPU time available to handle kernel and user processes diminishes as packet arrival rate becomes too large.

Figure 3.13 shows the interrupt overhead for different T_{ISR}'s. Notice that, if T_{ISR} time increases the interrupt overhead increases. Figure 3.14 illustrates the relation between the CPU availability and CPU utilization due to interrupt handling. At lower arrival rate, the ISR overhead is not significant since the system has much time to process packets. When the interarrival time is equal to T_{ISR}, i.e. $1/\lambda = T_{ISR}$, then the CPU availability and CPU utilization due to ISR handling are equal (50%). After this point, the CPU consumes most of its time to handle ISR than to process a task.

Thus, by using Equations (3-22) and (3-24), the mean effective service rate can be expressed as

$$\mu' = \mu\left(\frac{r}{\lambda + r}\right). \qquad (3\text{-}26)$$

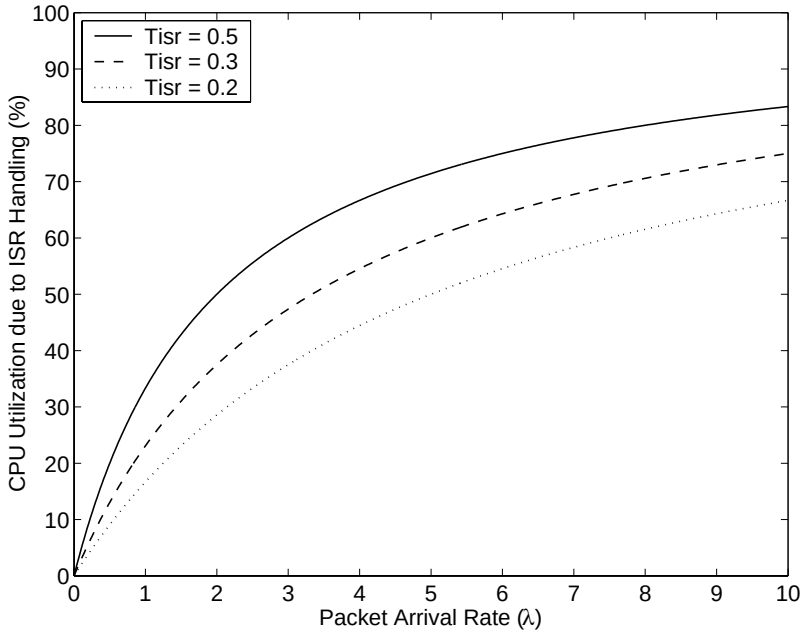


Figure 3.13: %CPU utilization vs. packet arrival rate

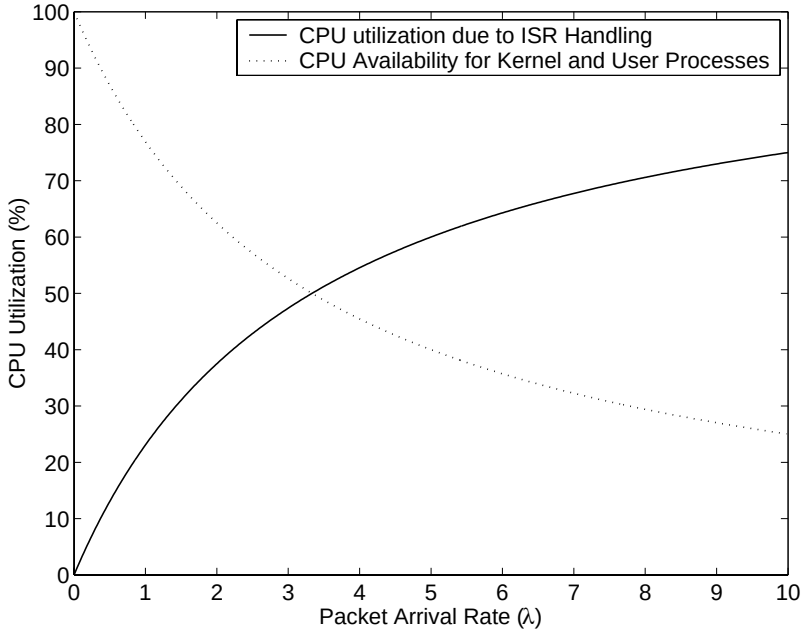


Figure 3.14: Relation between CPU availability and

CPU utilization due to ISR handling

3.3.2.1.1 Performance Metrics

It is to be noted from Equation (3-22) that the mean effective service rate μ' is exponential. Therefore, we can model the system as M/M/1/B queue as in the case for the Ideal system. However, the mean service rate μ will be replaced by the mean effective service rate μ'. Hence, system throughput, latency, and CPU availability are expressed by Equations (3-8), (3-10), and (3-11) respectively.

Therefore, the overall system power can be expressed as

$$P(\lambda) = \frac{\lambda^a (\mu r - \lambda(\lambda + r))^{b+c}}{(\mu r)^b (\lambda + r)^c}. \qquad (3\text{-}27)$$

Finding the optimal operating point for Equation (3-27) is quit complicated, but we can obtain the optimal point when *a*, *b*, and *c* are equal to 1. Thus, making all the tunable parameters equal to 1 and taking the derivative of Equation (3-27) with respect to λ, we get

$$\frac{dP}{d\lambda} = -4\lambda + \frac{\lambda^2(4\lambda + 3r)}{\mu r} + \frac{\mu r^2}{(\lambda + r)^2}.$$

Let $dP/d\lambda = 0$, the resultant equation has two positive real solutions. One of them represents the knee point of the power function (local maximum) and the other represents the cliff point where the power is zero (local minimum). Thus, the optimal operating point is given by

$$\lambda = \frac{1}{12\sigma}(\sigma^{\frac{2}{3}} + 13r^2 + 7r\sigma^{\frac{1}{3}}), \qquad (3\text{-}28)$$

where $\sigma = 35r^3 + 216\mu r^2 + 6\sqrt{3}\sqrt{r^4(432\mu^2 + 140\mu r - 9r^2}}$.

The cliff point is given by

$$\lambda = \frac{1}{2}\left(-r + \sqrt{r^2 + 4\mu r}\right). \qquad (3\text{-}29)$$

The stability condition for the system can be expressed as

$$\rho < 1 \quad \text{or} \quad \lambda < \mu\left(\frac{r}{\lambda + r}\right).$$

Solving for λ, we get

$$\lambda(\lambda + r) < \mu r \quad \Rightarrow \quad \lambda^2 + r\lambda - \mu r < 0.$$

The roots of the quadratic equation $\lambda^2 + r\lambda - \mu r = 0$ are

$$\lambda = \frac{-r \pm \sqrt{r^2 + 4\mu r}}{2} = \frac{-r \pm r\sqrt{1 + 4\dfrac{\mu}{r}}}{2}.$$

Since the term under the square root is always greater than one then the negative sign is neglected. Therefore, the system will be stable whenever

$$\lambda < \frac{r}{2}\left(\sqrt{1 + 4\frac{\mu}{r}} - 1\right). \qquad (3\text{-}30)$$

Clearly, this is the same equation as Equation (3-29).

Special Case. We consider a special case when interrupt handling is ignored (T_{ISR} = 0) in order to validate our mathematical equations. In this situation, when T_{ISR} = 0, $r \to \infty$. We prove that Equations (3-26) and (3-30) yield the same equations of the ideal system model, i.e., M/M/1/B queueing system, as follows:

For finding mean effective service rate of Equation (3-26),

$$\mu' = \lim_{r\to\infty}\left(\frac{\mu r}{\lambda + r}\right) = \lim_{r\to\infty}\left(\frac{\mu}{\lambda/r + 1}\right) = \mu .$$

For finding λ for stability condition of Equation (3-30),

$$\lambda = \lim_{r\to\infty}\left(\frac{r}{2}\sqrt{1+4\frac{\mu}{r}} - \frac{r}{2}\right) = \lim_{r\to\infty}\left(\frac{\sqrt{1+4\frac{\mu}{r}} - 1}{\frac{2}{r}}\right).$$

Applying L'Hopital Rule, we get

$$\lambda = \lim_{r\to\infty}\left(\frac{-2\mu}{r^2\sqrt{1+4\mu/r}} \Big/ \frac{-2}{r^2}\right) = \lim_{r\to\infty}\left(\frac{\mu}{\sqrt{1+2\mu/r}}\right) = \mu .$$

3.3.2.1.2 Numerical Results

We now show some numerical results of our analytical model to study the behavior of the system and the impact of interrupts on system performance. As we did before, we fix μ to 1 and *B* to a size of 100.

We first examine the system throughput as a function of traffic intensity, ρ. We study this relation with four T_{ISR} time units 0.2, 0.3, and 0.5.

Figure 3.15 depicts the impact of high and low traffic intensity on system throughput. The figure shows the system throughput for three cases of T_{ISR} 0.2, 0.3, and 0.5. It is noted that as the interrupt overhead increases (increasing the value of T_{ISR}), the system throughput is worsen and the livelock phenomenon occurs earlier.

Figure 3.15 also shows the cliff points for the system throughput. As previously defined, the cliff points are those points where system throughput starts falling to zero as the system load increases. As shown, the cliff points in terms of traffic intensity ρ for

T_{ISR} of 0.2, 0.3, and 0.5 are 0.85, 0.81, and 0.73, respectively. Since we are fixing μ to 1, the cliff points are the same for the system throughput, traffic intensity, and packet arrival rate. These points match exactly the points derived by Equation (3-30) for finding the stability condition.

Figure 3.16 illustrates the relation between packet latency and traffic intensity for the same system parameter values considered for system throughput. It is shown that the latency for the Ideal system is the least and it is the worst when T_{ISR} takes the largest value of 0.5.

Figure 3.17 illustrates the relation between CPU availability for user processes and traffic intensity for the same system parameter values. It is shown that as interrupt overhead is increased, the CPU availability is worsened.

The impact of low and high traffic intensity of overall system power is shown in Figure 3.18. In the Ideal system, the maximum overall power is when $\rho = 0.33$. However, the maximum overall system power decreases with different values of T_{ISR}, giving the least value for T_{ISR} = 0.5. In addition, the figure shows that the maximum power point for the system for T_{ISR} of 0.2, 0.3, and 0.5 are for λ of 0.292, 0.277, and 0.253, respectively. These points match also exactly with the points we derived by Equation (3-28) for finding λ that gives the maximum power point.

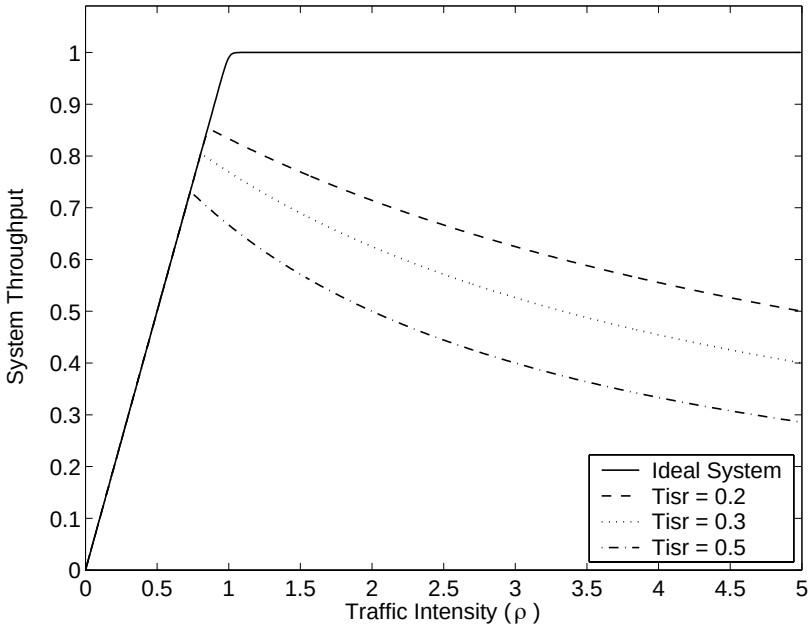


Figure 3.15: System throughput for Traditional scheme based on Effective Service Time technique

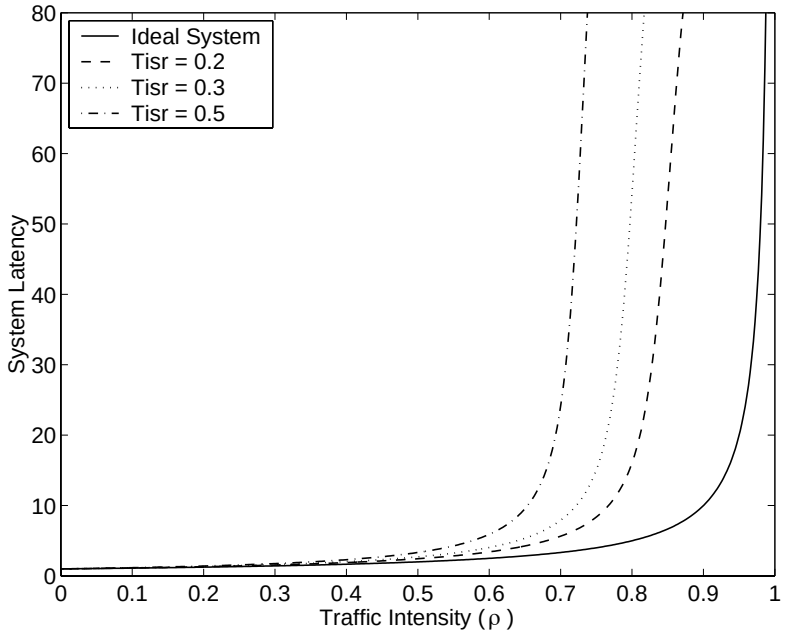


Figure 3.16: System latency for Traditional scheme based on Effective Service Time technique

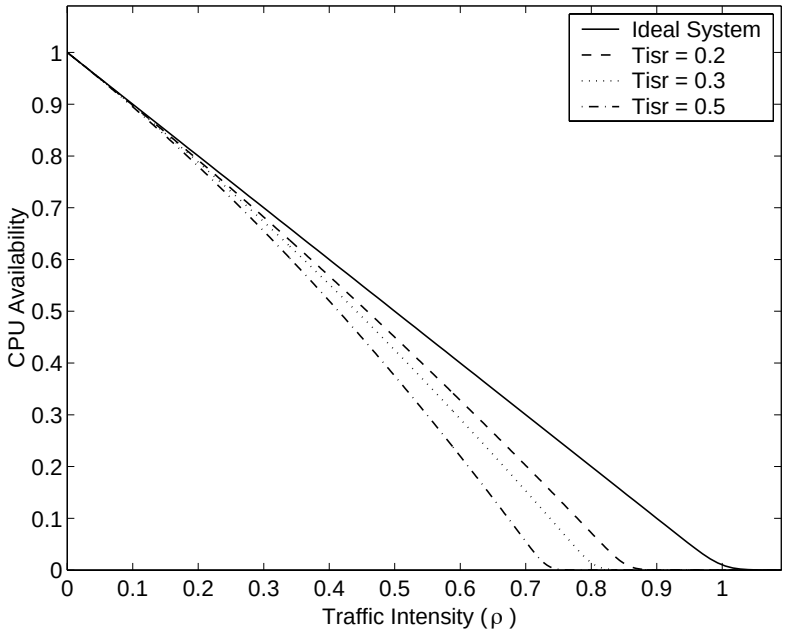


Figure 3.17: CPU availability for Traditional scheme

based on Effective Service Time technique

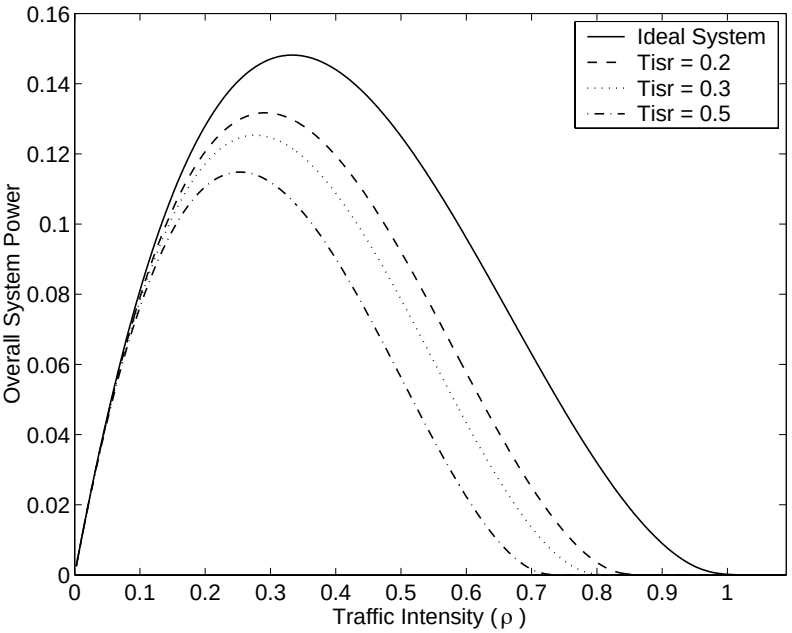


Figure 3.18: Overall system power for Traditional scheme

based on Effective Service Time technique

3.3.2.2 Second Technique: Pure Markovian Chain

As opposed to first technique, the model we consider is one in which the server has two mean rates: ISR and protocol processing. Service times for ISR and packet processing are exponentially distributed with mean $1/r$ and $1/\mu$ respectively. If the server is processing a packet and a new packet arrives, then the server switches to ISR. While the server is executing an ISR and a new packet arrives, the server will remain in ISR without affecting ISR service time.

The described scenario can be modeled as a pure Markov chain with a state space $S = \{ (n,m), 0 \le n \le \infty, m \in \{0,1\} \}$, where n denotes the number of packets in the buffer and m denotes the type of service. 0 indicates protocol processing and 1 indicates ISR handling. The rate transition diagram is shown in Figure 3.19.

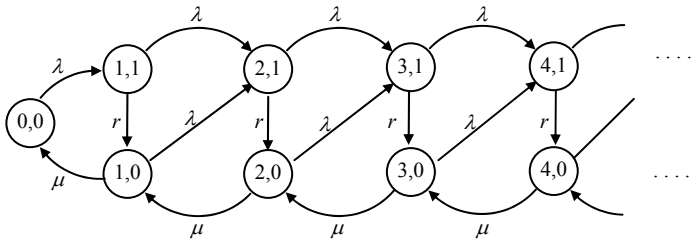


Figure 3.19: Rate transition diagram for traditional interrupt-driven system

Let $p_{n,m}$ be the steady-state probability where n denotes the number of packets in the system and m denotes the type of current service. A system of difference equations can be derived for the stationary probabilities as follows:

$$\left.\begin{aligned} 0 &= -\lambda p_{0,0} + \mu p_{1,0}\,, \\ 0 &= -(\lambda + r) p_{1,1} + \lambda p_{0,0}\,, \\ 0 &= -(\lambda + \mu) p_{n,0} + r p_{n,1} + \mu p_{n+1,0} \qquad n \geq 1\,, \\ 0 &= -(\lambda + r) p_{n,1} + \lambda p_{n-1,0} + \lambda p_{n-1,1} \qquad n \geq 2\,. \end{aligned}\right\} \quad (3\text{-}31)$$

The first two equations constitute the initial values. The last two equations constitute the system of difference equations. In order to solve this system of equations, we need to re-arrange them as follows:

$$p_{n+1,0} = \frac{\lambda + \mu}{\mu} p_{n,0} - \frac{r}{\mu} p_{n,1} \qquad n \geq 1\,,$$

$$p_{n+1,1} = \frac{\lambda}{\lambda + r} p_{n,0} + \frac{\lambda}{\lambda + r} p_{n,1} \qquad n \geq 1\,.$$

These equations can be written in the vector form as follows:

$$p(n+1) = A\, p(n)\,,$$

where

$$A = \begin{bmatrix} \dfrac{\lambda + \mu}{\mu} & -\dfrac{r}{\mu} \\ \dfrac{\lambda}{\lambda + r} & \dfrac{\lambda}{\lambda + r} \end{bmatrix},$$

$$p(n) = \begin{bmatrix} p_{n,0} \\ \\ p_{n,1} \end{bmatrix}, \text{ and } p(n+1) = \begin{bmatrix} p_{n+1,0} \\ \\ p_{n+1,1} \end{bmatrix}.$$

Therefore, our equations have been nicely converted to a system of first order difference equation, in which we can apply *Putzer algorithm* to obtain the solution [ELAY96].

Before we proceed for solution, let us denote $\alpha = \lambda / \mu$, and $\beta = \lambda /(\lambda + r)$. Then, matrix A can be rewritten as follows:

$$A = \begin{bmatrix} \alpha + 1 & -\alpha(1-\beta)/\beta \\ \beta & \beta \end{bmatrix}.$$

The eigenvalues of matrix A can be obtained by solving the characteristic equation $\det(A - z\,I) = 0$ where z is the eigenvalue, and I is the identity matrix. Now

$$\det(A - zI) = \det\begin{bmatrix} \alpha + 1 - z & -\alpha(1-\beta)/\beta \\ \beta & \beta - z \end{bmatrix} = (1-z)(z-\alpha-\beta) = 0.$$

Hence, the eigenvalues of matrix A are $z_1 = 1$ and $z_2 = \alpha + \beta$.

So, according to *Putzer Algorithm*,

$$M(0) = I, \quad \text{and} \quad M(1) = A - z_1 I = \begin{bmatrix} \alpha & -\alpha(1-\beta)/\beta \\ \beta & \beta - 1 \end{bmatrix}.$$

Then,

$$u_1(n) = 1^n = 1,$$

and

$$u_2(n) = \sum_{i=0}^{n-1} (\alpha+\beta)^{n-1-i}(1^i) = \frac{1-(\alpha+\beta)^n}{1-(\alpha+\beta)}.$$

Finally, we have

$$A^n = u_1(n)\times M(0) \quad + \quad u_2(n)\times M(1)$$

$$= \begin{bmatrix} \dfrac{1-\beta-\alpha(\alpha+\beta)^n}{1-(\alpha+\beta)} & \dfrac{\alpha(1-\beta)(1-(\alpha+\beta)^n)}{\beta(1-(\alpha+\beta))} \\ \dfrac{\beta(1-(\alpha+\beta)^n)}{1-(\alpha+\beta)} & \dfrac{-\alpha+(1-\beta)(\alpha+\beta)^n}{1-(\alpha+\beta)} \end{bmatrix}.$$

The solution of the difference equation is given by

$$p(n+1) = A^n p(1) = \begin{bmatrix} \dfrac{1-\beta-\alpha(\alpha+\beta)^n}{1-(\alpha+\beta)} & \dfrac{\alpha(1-\beta)(1-(\alpha+\beta)^n)}{\beta(1-(\alpha+\beta))} \\ \dfrac{\beta(1-(\alpha+\beta)^n)}{1-(\alpha+\beta)} & \dfrac{-\alpha+(1-\beta)(\alpha+\beta)^n}{1-(\alpha+\beta)} \end{bmatrix} \times \begin{bmatrix} \alpha\, p_{0,0} \\ \beta\, p_{0,0} \end{bmatrix}$$

$$= \begin{bmatrix} \dfrac{1-\beta-\alpha(\alpha+\beta)^n}{1-(\alpha+\beta)} \times \alpha\, p_{0,0} + \dfrac{\alpha(1-\beta)(1-(\alpha+\beta)^n)}{\beta(1-(\alpha+\beta))} \times \beta\, p_{0,0} \\ \dfrac{\beta(1-(\alpha+\beta)^n)}{1-(\alpha+\beta)} \times \alpha\, p_{0,0} + \dfrac{-\alpha+(1-\beta)(\alpha+\beta)^n}{1-(\alpha+\beta)} \times \beta\, p_{0,0} \end{bmatrix}$$

The solution can be nicely simplified as

$$p_{n,0} = \alpha\, p_{0,0}(\alpha+\beta)^{n-1}$$

$$\text{n} \geq 1 \qquad \text{(3-32)}$$

$$p_{n,1} = \beta\, p_{0,0}(\alpha+\beta)^{n-1}$$

To get $p_{0,0}$, we utilize the fact the probabilities must sum to 1 and it follows that

$$\sum_{n=1}^{\infty} p_{n,0} + \sum_{n=1}^{\infty} p_{n,1} + p_{0,0} = 1,$$

$$p_{0,0}\,\alpha \sum_{n=1}^{\infty}(\alpha+\beta)^{n-1} + p_{0,0}\,\beta \sum_{n=1}^{\infty}(\alpha+\beta)^{n-1} + p_{0,0} = 1.$$

Therefore

$$p_{0,0} = \left[1 + (\alpha + \beta)\sum_{n=1}^{\infty}(\alpha + \beta)^{n-1}\right]^{-1}.$$

Now $\sum_{n=1}^{\infty}(\alpha + \beta)^{n-1}$ is geometric series and converges if and only if $(\alpha + \beta) < 1$. Thus for the existence of a steady-state solution, $\rho = (\alpha + \beta)$ must be less that 1. Then, we have

$$p_{0,0} = \left[1 + \frac{\alpha + \beta}{1 - (\alpha + \beta)}\right]^{-1} = 1 - (\alpha + \beta) = 1 - \rho .$$

where $\rho = (\alpha + \beta)$, or equivalently, $\rho = \lambda / \mu + \lambda / (\lambda + r)$.

Thus the full steady-state solution for our system is the geometric probability functions

$$\left.\begin{aligned} p_{0,0} &= 1 - \rho \\ p_{n,0} &= \alpha(1 - \rho)\rho^{n-1} \\ p_{n,1} &= \beta(1 - \rho)\rho^{n-1} \end{aligned}\right\} \quad n \geq 1 \qquad (3\text{-}33)$$

where $\rho = \alpha + \beta$, $\alpha = \lambda / \mu$, and $\beta = \lambda /(\lambda + r)$.

Notice that the server utilization ρ consists of two terms. The first term α is the server utilization due to protocol processing and the second term β is the server utilization due to ISR handling. Note that β was expressed in Equation (3-25) for determining the CPU usage by ISR handling.

We consider a special case when interrupt overhead is ignored in order to validate our mathematical model. When $r \to \infty$, then $\beta \to 0$ and $\rho \to \lambda / \mu$.

In order to study the system performance, we have to measure the impact of low and high traffic intensity on system performance, i.e. when $\rho > 1$. To do this, we have

to model the system as a finite buffer of size B. In this situation, we ends up with two possible network design solutions. In the first solution, a new incoming packet will generate an interrupt, in spite of buffer availability. In the second solution, packets will be dropped when the buffer becomes full without generating interrupts.

Note that modeling the system as a finite buffer will yield the same equations described in Equations (3-32). The only change is the boundary probabilities $p_{0,0}$, $p_{B,0}$, and $p_{B,1}$. We next find these probabilities by considering the two different solutions.

3.3.2.2.1 Pure Markovian Model: First Solution

Figure 3.20 shows the rate-transition diagram for the first solution in which any packet will introduce an interrupt even if the buffer is full.

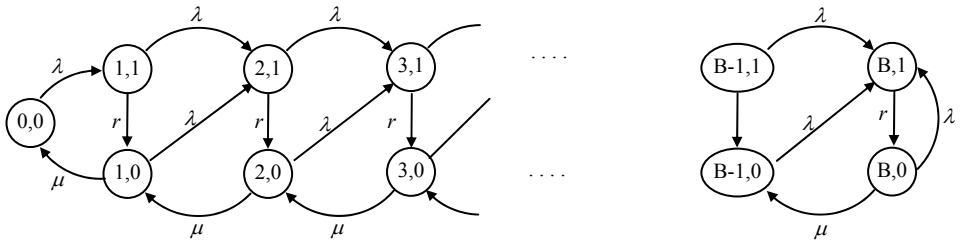


Figure 3.20: Rate-transition diagram for modeling first solution

The boundary probabilities at state (B, m) are

$$-(\lambda+\mu)p_{B,0}+r\,p_{B,1}=0, \qquad (3\text{-}34)$$

$$-r\,p_{B,1}+\lambda\,p_{B-1,1}+\lambda\,p_{B-1,0}+\lambda\,p_{B,0}=0. \qquad (3\text{-}35)$$

Substitute Equation (3-35) into (3-34) , we have

$$-(\lambda+\mu)p_{B,0}+\lambda p_{B-1,1}+\lambda p_{B-1,0}+\lambda p_{B,0}=0,$$

$$\mu p_{B,0}=\lambda(p_{B-1,0}+p_{B-1,1}),$$

$$p_{B,0}=\frac{\lambda}{\mu}(p_{B-1,0}+p_{B-1,1}).$$

Use Equations (3-32) to obtain $p_{B-1,0}$ and $p_{B-1,1}$, and then substitute them into the above equation. We get

$$p_{B,0}=\alpha\, p_{0,0}\,(\alpha+\beta)^{B-1}. \qquad (3\text{-}36)$$

Now substitute Equation (3-36) into Equation (3-34), we have

$$p_{B,1}=\frac{\lambda}{r}\,p_{0,0}\,(\alpha+1)(\alpha+\beta)^{B-1}. \qquad (3\text{-}37)$$

Since the summation of all probabilities is equal to 1, we get

$$p_{0,0}+\sum_{n=1}^{B-1}(p_{n,0}+p_{n,1})+p_{B,0}+p_{B,1}=1,$$

$$p_{0,0}+p_{0,0}\sum_{n=1}^{B-1}(\alpha+\beta)^{n}+p_{0,0}(\alpha+\frac{\lambda}{r}(\alpha+1))(\alpha+\beta)^{B-1}=1.$$

Therefore,

$$p_{0,0}=\left[1+\frac{(\alpha+\beta)-(\alpha+\beta)^{B}}{1-(\alpha+\beta)}+(\alpha+\frac{\lambda}{r}(\alpha+1))(\alpha+\beta)^{B-1}\right]^{-1}.$$

Now, let $\rho=\alpha+\beta$ and $\alpha+\lambda/r\cdot(\alpha+1)=\rho/(1-\beta)$, then

$$p_{0,0} = \frac{1-\rho}{1-\frac{\alpha}{1-\beta}\rho^{B}}. \qquad (3\text{-}38)$$

3.3.2.2.2 Pure Markovian Model: Second Solution

Figure 3.21 shows the rate-transition diagram for the second solution in which packets will be dropped when the buffer is full without generating interrupts.

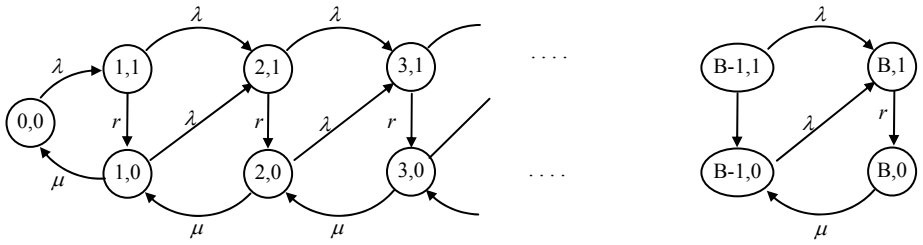


Figure 3.21: Rate-transition diagram for modeling second solution

The boundary probabilities at state (B, m) are

$$-\mu p_{B,0} + r p_{B,1} = 0, \qquad (3\text{-}39)$$

$$-r p_{B,1} + \lambda p_{B-1,1} + \lambda p_{B-1,0} = 0. \qquad (3\text{-}40)$$

Substitute Equation (3-39) into Equation (3-40), we get

$$-\mu p_{B,0} + \lambda p_{B-1,1} + \lambda p_{B-1,0} = 0,$$

$$\mu p_{B,0} = \lambda(p_{B-1,1} + p_{B-1,0}).$$

Use Equations (3-32) to obtain $p_{B-1,0}$ and $p_{B-1,1}$, and then substitute them into above equation. We get

$$p_{B,0} = \alpha p_{0,0} (\alpha + \beta)^{B-1}. \quad (3\text{-}41)$$

Notice that, the $p_{B,0}$ is similar for both cases.

Now substitute Equation (3-41) into Equation (3-39), we get

$$p_{B,1} = \frac{\mu}{r} \alpha p_{0,0} (\alpha + \beta)^{B-1}. \quad (3\text{-}42)$$

Again, we apply the boundary condition that the summation of all probabilities is equal to 1, we get

$$p_{0,0} + \sum_{n=1}^{B-1} (p_{n,0} + p_{n,1}) + p_{B,0} + p_{B,1} = 1,$$

$$p_{0,0} + p_{0,0} \sum_{n=1}^{B-1} (\alpha + \beta)^n + \alpha p_{0,0} (1 + \frac{\mu}{r})(\alpha + \beta)^{B-1} = 1.$$

Therefore,

$$p_{0,0} = \left[1 + \frac{(\alpha + \beta) - (\alpha + \beta)^B}{1 - (\alpha + \beta)} + \alpha (1 + \frac{\mu}{r})(\alpha + \beta)^{B-1} \right]^{-1}.$$

Now, let $\alpha(1 + \mu / r) = \alpha + \beta/(1 - \beta)$, then

$$p_{0,0} = \frac{1 - \rho}{1 - \alpha \left(1 + \frac{\beta}{\rho(1 - \beta)} \right) \rho^B}. \quad (3\text{-}43)$$

3.3.2.2.3 Performance Metrics

1. Throughput. Since the system throughput is rate at which packets are successfully leave the system, then throughput can be expressed as

$$\gamma = \mu \sum_{n=1}^{B} p_{n,0} = \mu \alpha \, p_{0,0} \sum_{n=1}^{B} \rho^{n-1} = \mu \alpha \, p_{0,0} \times \frac{1-\rho^{B}}{1-\rho} . \qquad (3\text{-}44)$$

2. Latency. It is suitable to calculate the mean response time when B approaches to infinity. Therefore, the expected number of packets in the system is:

$$E(n) = \sum_{n=1}^{\infty} n (p_{n,0} + p_{n,1}) = \sum_{n=1}^{\infty} n \times \left[\alpha (1-\rho) \rho^{n-1} + \beta (1-\rho) \rho^{n-1} \right]$$

$$= (\alpha + \beta)(1-\rho) \sum_{n=1}^{\infty} \rho^{n-1} = \rho (1-\rho) \left(\frac{1}{1-\rho} \right)^{2} = \frac{\rho}{1-\rho} .$$

Thus, mean response time is expressed as Equation (3-4).

3. CPU availability. CPU availability is expressed as

$$V = p_{0,0} . \qquad (3\text{-}45)$$

4. Overall system power. It is suitable to express system throughput, latency and CPU availability as infinite system states. Thus

$$\gamma = \mu \sum_{n=1}^{\infty} p_{n,0} = \mu \alpha (1-\rho) \sum_{n=1}^{\infty} \rho^{n-1} = \mu \alpha = \lambda ,$$

$$R(\lambda) = \frac{\rho}{\lambda (1-\rho)} = \frac{\lambda + \mu + r}{\mu r - \lambda (\lambda + r)} ,$$

and

$$V(\lambda) = 1 - \rho = \frac{\mu r - \lambda (\lambda + r)}{\mu \ (\lambda + r)} .$$

Thus, the overall power is expressed as

$$P(\lambda) = \frac{\lambda^{a} \left(\mu r - \lambda (\lambda + r) \right)^{b+c}}{\mu^{b} \ (\lambda + r)^{b} (\lambda + \mu + r)^{c}} . \qquad (3\text{-}46)$$

Hence, By making all the tunable parameters equal to 1 and taking the derivative of Equation (3-46) with respect to λ, we get

$$\frac{dP}{d\lambda} = \frac{r^3}{(\lambda + r)^2} + \frac{3\lambda^2}{\mu} + \lambda\left(\frac{\lambda + r}{\lambda + \mu + r}\right)^2 - (3\lambda + r)\cdot\left(\frac{\lambda + r}{\lambda + \mu + r}\right).$$

Solving for $dP/d\lambda = 0$ is quit complicated. Numerical methods are needed to obtain the maximum power point.

5. Stability condition. We have

$$\rho < 1 \quad \text{or} \quad \lambda/\mu + \lambda/(\lambda + \mathrm{r}) < 1.$$

Solving for λ, we get

$$\frac{\lambda(\lambda + \mathrm{r}) + \lambda\mu}{\mu(\lambda + \mathrm{r})} < 1,$$

$$\lambda(\lambda + \mathrm{r}) + \lambda\mu < \mu(\lambda + \mathrm{r}),$$

$$\lambda(\lambda + \mathrm{r}) + \lambda\mu - \mu(\lambda + \mathrm{r}) < 0,$$

$$\lambda^2 + r\lambda - \mu r < 0.$$

This is the same quadratic equation we have obtained in section 3.3.2.1. Therefore, the stability condition is expressed as Equation (3-30).

3.3.2.2.4 Numerical Results

Now, we show some numerical results of our analytical models to examine the system behavior and the impact of interrupts on system performance. As in section 3.2.2, we fix μ to 1 and B to a size of 100.

Figure 3.22 depicts the system throughput for analytic model of the first solution. We see that Figure 3.22 is quite similar to Figure 3.15 of our first analytic model.

However, Figure 3.23 which represents the second solution shows an improved behavior of system throughput. This is because the dropped packet will not introduce an interrupt since they are dropped at early stage. At high arrival rate, the probability of dropping packets is very high and therefore preventing the interrupt generation for those dropped packets will significantly improve the system throughput.

However, both models (i.e. the first solution and the second solution) give same behavior in terms of system latency and CPU availability for user processes, as shown in Figure 3.24 and Figure 3.25.

Figure 3.26 depicts the overall system power of pure Markovian model. The figure shows the same behavior as Figure 3.18, in which the maximum overall system power decreases with different values of T_{ISR}. In addition, the figure shows that the maximum power point for the system for T_{ISR} of 0.2, 0.3, and 0.5 are for λ of 0.283, 0.266, and 0.241, respectively.

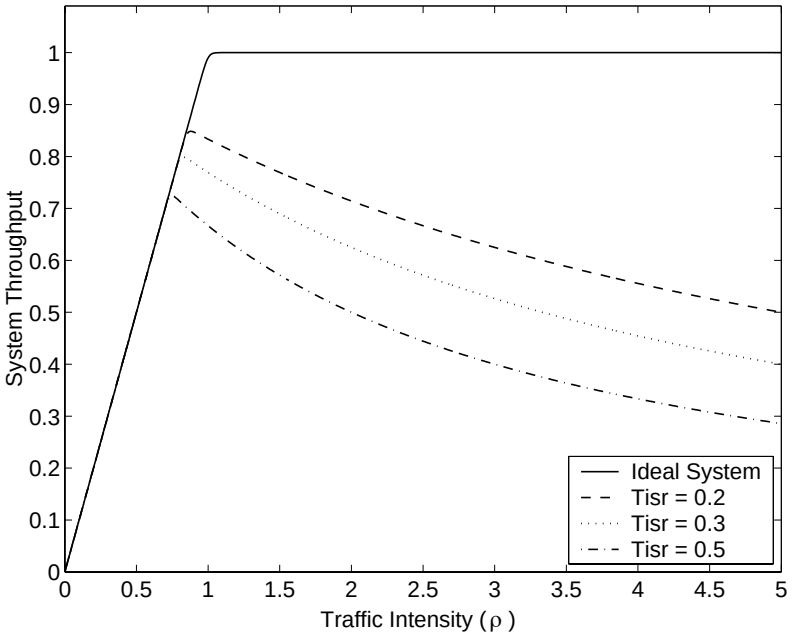


Figure 3.22: System throughput for Traditional scheme based on pure Markovian model – First solution

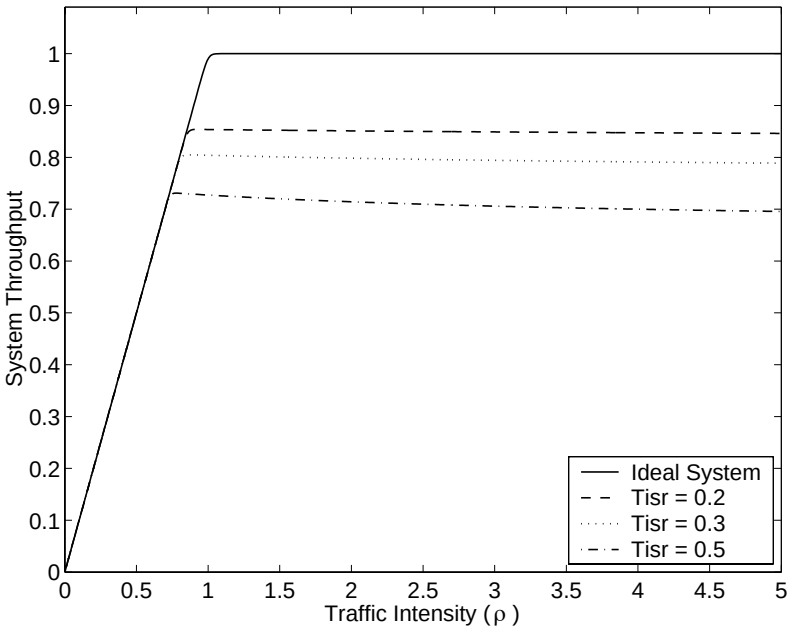


Figure 3.23: System throughput for Traditional scheme based on pure Markovian model – Second solution

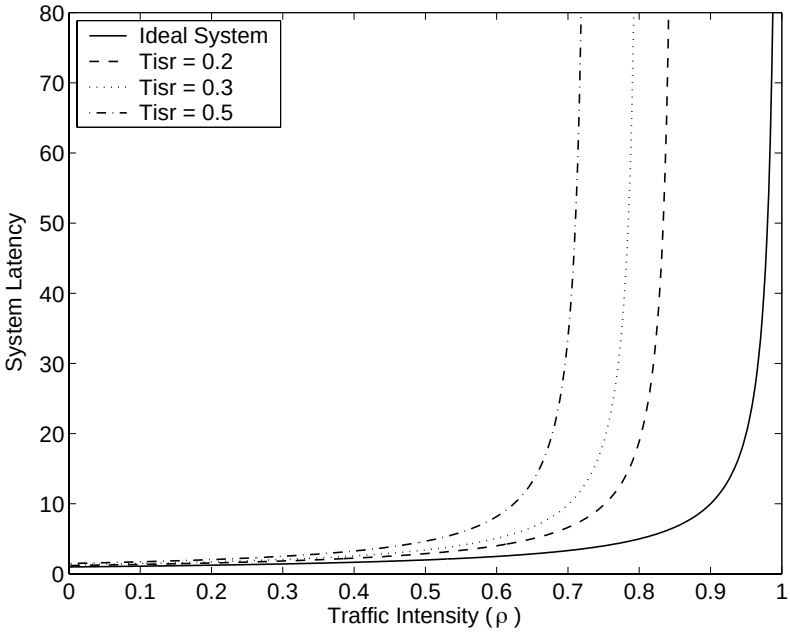


Figure 3.24: System latency for Traditional scheme based on pure Markovian model

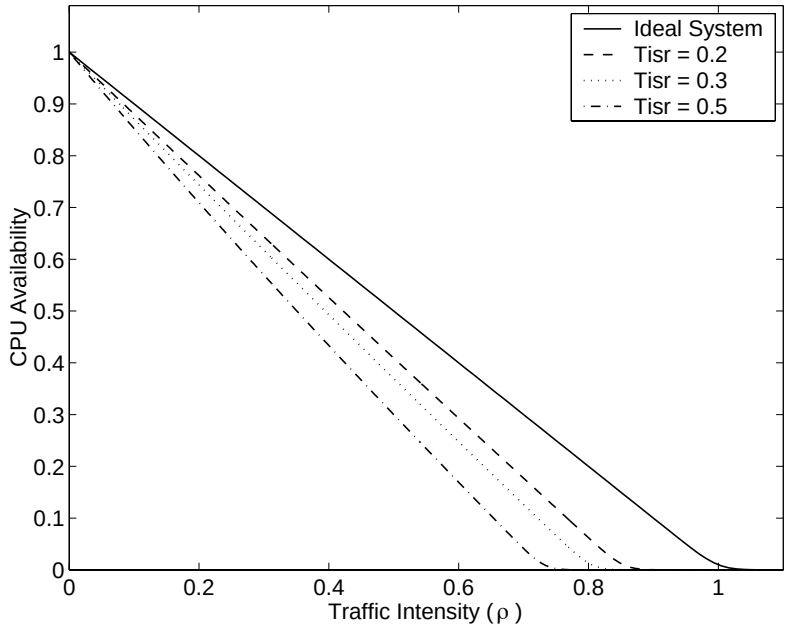


Figure 3.25: CPU availability for Traditional scheme based on pure Markovian model

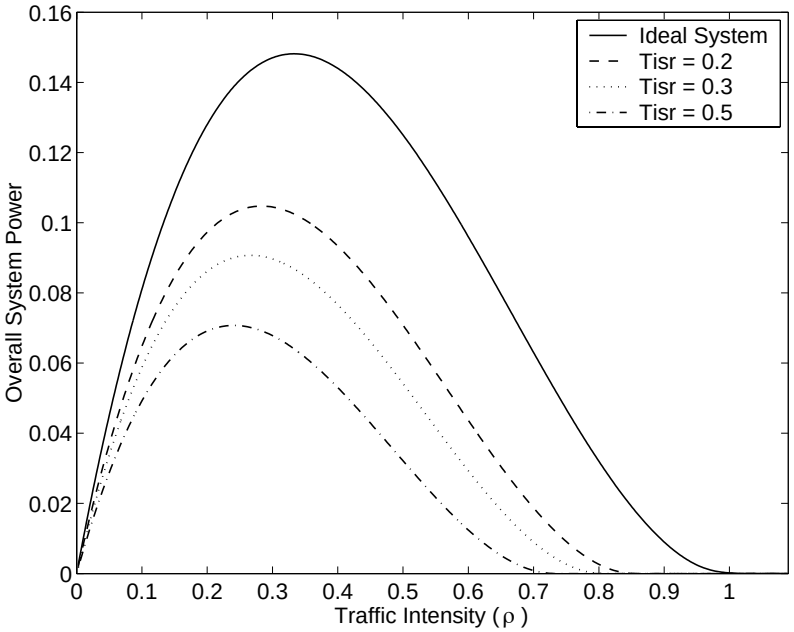


Figure 3.26: Overall system power for Traditional scheme based on pure Markovian model

3.3.2.3 Comparison of Two Models

This section compares between the first analytic model based on effective service time and the second analytic model based on pure Markovian model. The comparison is shown in Figure 3.27, Figure 3.28, and Figure 3.29. In all of these figures, we fix T_{ISR} to 0.3.

Figure 3.27 illustrates the comparison between the two models in terms of system throughput. We used the second solution of pure Markovian model for comparison. Notice that the figure shows only a single graph. This means that the two models produce exact results for system throughput.

Figure 3.28 shows the difference between the two models in terms of system latency. We notice that the two models are not quite different; as a matter of fact they are approximately the same.

Figure 3.29 compares between the two models in terms of CPU availability. It is noted that the two models are not exact as in the case in system throughput. But, the two models have the same behavior for modeling CPU availability in interrupt-driven system.

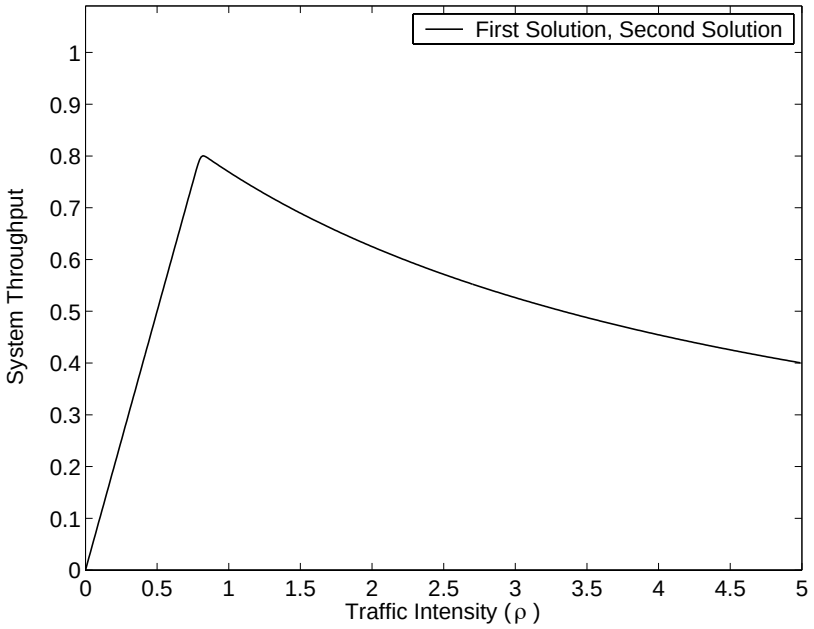


Figure 3.27: System throughput for both first and second analytic models

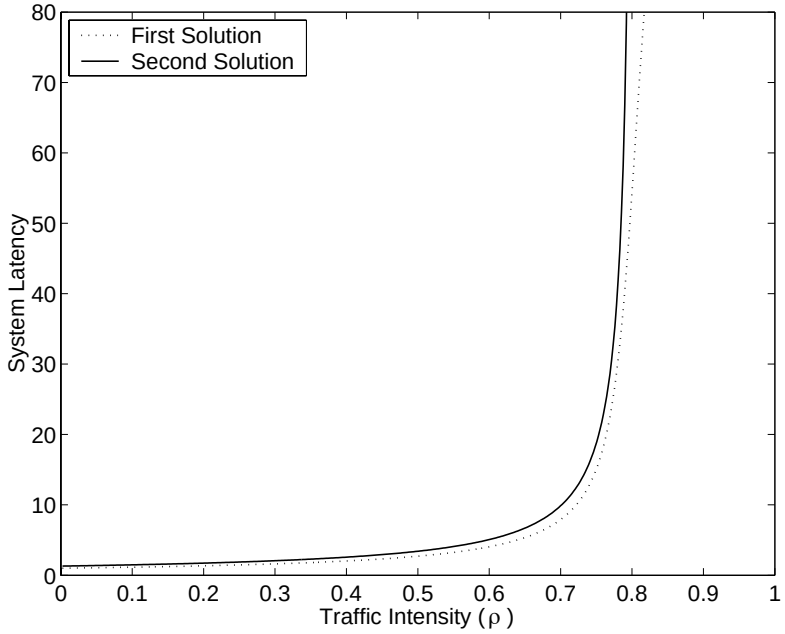


Figure 3.28: CPU availability for both first and second analytic models

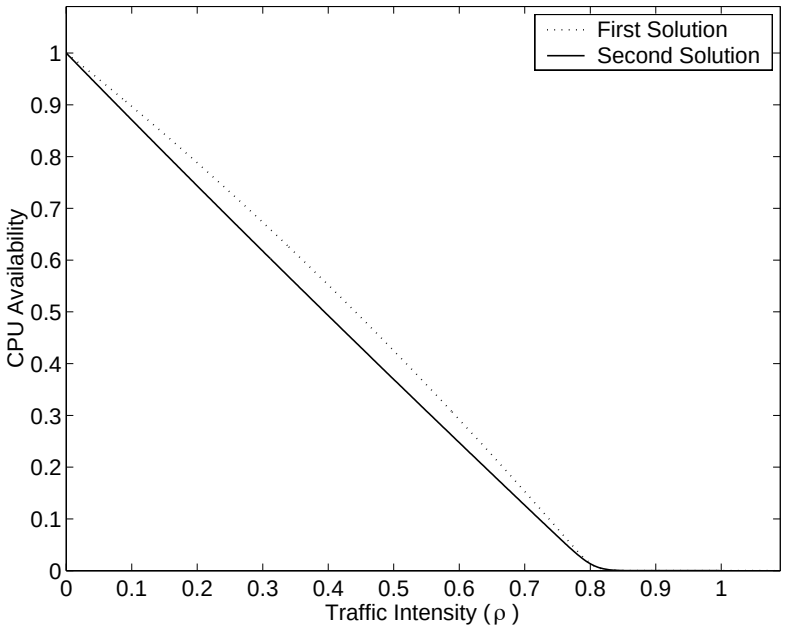


Figure 3.29: System latency for both first and second analytic models

3.4 Interrupt Coalescing Model

We mentioned in Chapter 2 that the interrupt coalescing is a mechanism to mitigate interrupt overhead by generating one single interrupt for multiple interrupts. As it was noted, there are two different schemes for interrupt coalescing. The first scheme generates an interrupt only if it receives a predefined number of packets. The second scheme generates an interrupt after a predefined time period. The timer is only triggered when receiving a new packet. When the time period is expired, the timer will be stopped and the NIC will issue a single interrupt indicating the reception of all packets received during the time period.

Figure 3.30 illustrates two timelines where packets arrive exponentially with mean $1/\lambda$ times unit. Figure 3.30 (a) represents the first approach where the number of packets per interrupt is equal to two. Therefore, the interrupt rate is $\lambda/2$. Figure 3.30 (b) represents the second approach where T is the timer duration. If $T < 1/\lambda$ then an interrupt will be generated before a new packet arrives. Therefore, the number of packets per T is one and the interrupt rate is λ. If $1/\lambda < T < 2/\lambda$, as shown in Figure 3.30 (b), then the number of packets per T is two and the interrupt rate will be $\lambda/2$.

We want to show that the predefined time T can be expressed as the number of packets per interrupt.

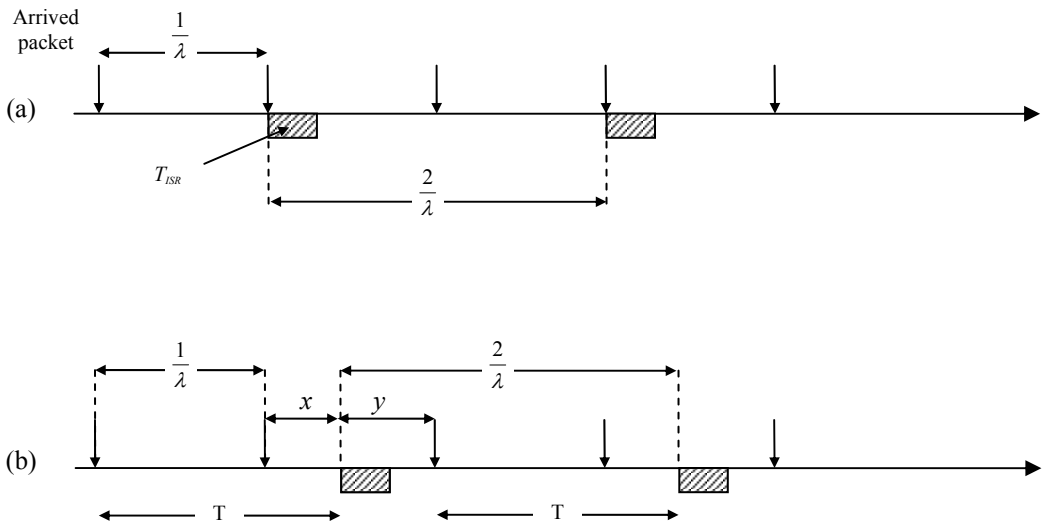


Figure 3.30: Timeline represents interrupt coalescing schemes

Let τ be the defined number of packets per *T*, then τ can be expressed as

$$\tau = \left\lceil \frac{T}{1/\lambda} \right\rceil = \lceil \lambda T \rceil. \qquad (3\text{-}47)$$

Let x be defined as the time between the last packet received before an interrupt generation and time *T*, as shown in Figure 3.30, Let y be defined as the time between *T* and the next incoming packet. Clearly, y can be expressed as $1/\lambda - x$. Thus, the time between two successive interrupts is

$$y + (\tau - 1) \cdot \frac{1}{\lambda} + x = \frac{1}{\lambda} - x + \frac{\tau}{\lambda} - \frac{1}{\lambda} + x = \frac{\tau}{\lambda}.$$

And hence, the interrupt frequency is λ / τ.

Therefore, τ specifies the number of packets per interrupt which is similar to the first scheme. In general, all interrupt coalescing schemes used in literature follow one scenario to generate interrupts and we can express the interrupt rate as

$$I_{freq} = \frac{1}{\max\left(\frac{1}{\lambda}, \frac{\tau}{\lambda}\right)}. \quad (3\text{-}48)$$

Equation (3-48) says that if $\tau \le 1$ then $I_{freq} = \lambda$, otherwise $I_{freq} = \lambda/\tau$.

3.4.1 Modeling CPU Usage

We implement the same idea of determining effective service time based on CPU availability for protocol processing as mentioned in section 3.3.2.1.

We use a Markov chain to model the CPU usage, as illustrated in Figure 3.31. The state space has states $(0,k)$ and states $(1,n)$. State $(0,k)$ with $0 \le k < \tau$ represents the state where the CPU is available for protocol processing. k denotes the number of packet arrivals that are being collected before generating an interrupt. State $(1,n)$ with $n \ge 0$ represents the state where the CPU is busy handling interrupts. n denotes the number of packet arrivals that are being batched or masked off during T_{ISR}. Note that when process in state (1,0), this means there are no interrupts being masked off and the CPU is handling a single interrupt.

When the system returns from ISR at state (1,0), this means the system will generate an interrupt after a batch of size τ. If the system returns from ISR at state (1,1), this means the system has already one packet waiting. Therefore, the system will

generate an interrupt after a batch of size τ-1. The system should return to state (0,1). Generally, if the system returns from ISR at state (1, n), then the system should return to state (0, n mod τ).

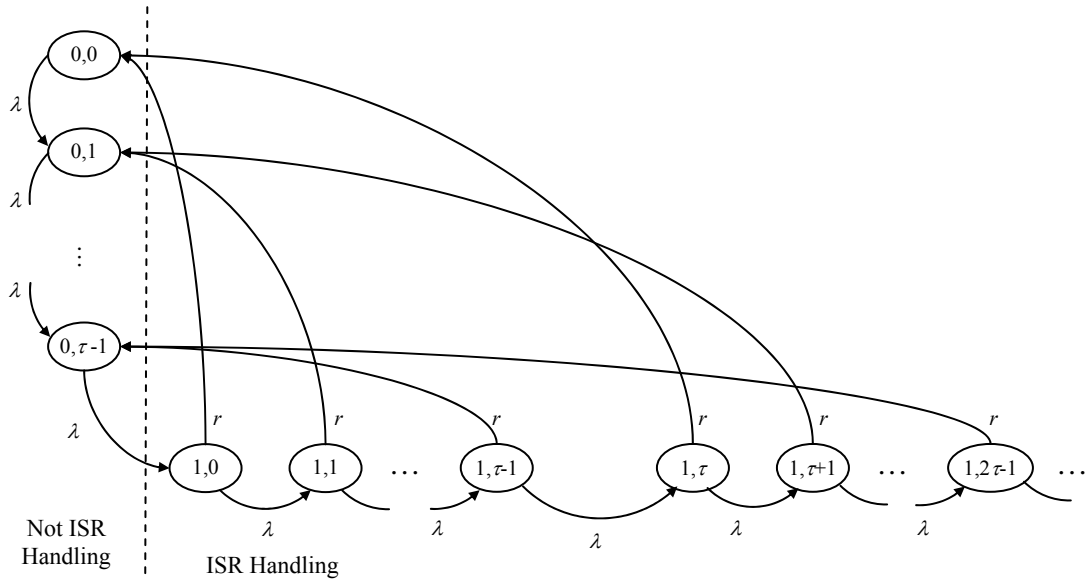


Figure 3.31: Modeling CPU usage for interrupt coalescing scheme

Using Equation (3-4), we have the following system of difference equations:

For states $(0,k)$, we have

$$\left.\begin{aligned}
&-\lambda p_{0,0} + r p_{1,0} + r p_{1,\tau} + r p_{1,2\tau} + r p_{1,3\tau} + \cdots = 0,\\
&-\lambda p_{0,1} + \lambda p_{0,0} + r p_{1,1} + r p_{1,\tau+1} + r p_{1,2\tau+1} + r p_{1,3\tau+1} + \cdots = 0,\\
&-\lambda p_{0,2} + \lambda p_{0,1} + r p_{1,2} + r p_{1,\tau+2} + r p_{1,2\tau+2} + r p_{1,3\tau+2} + \cdots = 0,\\
&\vdots\\
&-\lambda p_{0,k} + \lambda p_{0,k-1} + r p_{1,k} + r p_{1,\tau+k} + r p_{1,2\tau+k} + r p_{1,3\tau+k} + \cdots = 0,\\
&\text{where} \qquad 0 \le k \le \tau - 1.
\end{aligned}\right\} \quad (3\text{-}49)$$

For states $(1,n)$, we have

$$\left.\begin{aligned}
&-(\lambda+r)\,p_{1,0}+\lambda\,p_{0,\tau-1}=0\,,\\
&-(\lambda+r)\,p_{1,1}+\lambda\,p_{1,0}=0\,,\\
&-(\lambda+r)\,p_{1,2}+\lambda\,p_{1,1}=0\,,\\
&\vdots\\
&-(\lambda+r)\,p_{1,n}+\lambda\,p_{1,n-1}=0 \qquad n\ge 0\,.
\end{aligned}\right\} \quad (3\text{-}50)$$

Let $\beta=\lambda/(\lambda+r)$, then Equations (3-50) can be simplified as

$$p_{1,n}=\left(\frac{\lambda}{\lambda+r}\right)p_{1,n-1}=\beta^{n+1}\,p_{0,\tau-1} \qquad n\ge 0\,. \quad (3\text{-}51)$$

In order to solve Equations (3-49), we need to express each equation in (3-49) with respect to $p_{0,\tau-1}$. Then

$p_{0,0}$ can be expressed as

$$\begin{aligned}
\lambda\,p_{0,0} &= r\left(p_{1,0}+p_{1,\tau}+p_{1,2\tau}+p_{1,3\tau}+\cdots\right)\\
&= r\left(\sum_{n=0}^{\infty}p_{1,n\tau}\right)=r\left(\sum_{n=0}^{\infty}\beta^{n\tau+1}p_{0,\tau-1}\right)=r\left(\frac{\beta}{1-\beta^{\tau}}\right)p_{0,\tau-1}\,,
\end{aligned}$$

$$p_{0,0}=\frac{r}{\lambda}\left(\frac{\beta}{1-\beta^{\tau}}\right)p_{0,\tau-1}=\left(\frac{1-\beta}{\beta}\right)\cdot\left(\frac{\beta}{1-\beta^{\tau}}\right)p_{0,\tau-1}=\left(\frac{1-\beta}{1-\beta^{\tau}}\right)p_{0,\tau-1}\,.$$

$p_{0,1}$ can be expressed as

$$\begin{aligned}\lambda p_{0,1} &= \lambda p_{0,0} + r\left(p_{1,1} + p_{1,\tau+1} + p_{1,2\tau+1} + p_{1,3\tau+1} + \cdots\right) \\ &= \lambda p_{0,0} + r\left(\sum_{n=0}^{\infty} p_{1,n\tau+1}\right) = \lambda p_{0,0} + r\left(\sum_{n=0}^{\infty} \beta^{n\tau+2} p_{0,\tau-1}\right) \\ &= \lambda p_{0,0} + r\left(\frac{\beta^2}{1-\beta^{\tau}}\right) p_{0,\tau-1}\ ,\end{aligned}$$

$$\begin{aligned}p_{0,1} &= \left(\frac{1-\beta}{1-\beta^{\tau}}\right) p_{0,\tau-1} + \left(\frac{1-\beta}{\beta}\right) \cdot \left(\frac{\beta^2}{1-\beta^{\tau}}\right) p_{0,\tau-1} \\ &= \left(\frac{(1-\beta)+\beta(1-\beta)}{1-\beta^{\tau}}\right) p_{0,\tau-1} = \frac{(1-\beta)(1+\beta)}{1-\beta^{\tau}}\, p_{0,\tau-1}\,.\end{aligned}$$

Generally, $p_{0,k}$ can be expressed as

$$p_{0,k} = \frac{(1-\beta)(1+\beta^2+\beta^3+\cdots+\beta^k)}{1-\beta^{\tau}}\, p_{0,\tau-1} \qquad 0 \le k \le \tau - 1\,.$$

or

$$p_{0,k} = \frac{1-\beta^{k+1}}{1-\beta^{\tau}}\, p_{0,\tau-1}\,. \qquad (3\text{-}52)$$

To find the value of $p_{0,\tau-1}$, we utilizing the boundary condition that

$$\sum_{k=0}^{\tau-1} p_{0,k} + \sum_{n=0}^{\infty} p_{1,n} = 1\,.$$

This can be rewritten as

$$\sum_{k=0}^{\tau-2} p_{0,k} + p_{0,\tau-1} + \sum_{n=0}^{\infty} p_{1,n} = 1\,.$$

Since

$$\sum_{n=0}^{\infty} p_{1,n} = \sum_{n=0}^{\infty} \beta^{n+1} p_{0,\tau-1} = \left(\frac{\beta}{1-\beta}\right) p_{0,\tau-1}\,,$$

then, we have

$$p_{0,\tau-1}\left[\sum_{k=0}^{\tau-2}\left(\frac{1-\beta^{k+1}}{1-\beta^{\tau}}\right)+1+\frac{\beta}{1-\beta}\right]=1,$$

or

$$p_{0,\tau-1}=\left[\sum_{k=0}^{\tau-2}\left(\frac{1-\beta^{k+1}}{1-\beta^{\tau}}\right)+1+\frac{\beta}{1-\beta}\right]^{-1}=\frac{1-\beta^{\tau}}{\tau}. \qquad (3\text{-}53)$$

Substituting Equation (3-53) into Equations (3-51) and (3-52), we get

$$p_{0,n}=\left(\frac{1-\beta^{n+1}}{1-\beta^{\tau}}\right)\cdot\left(\frac{1-\beta^{\tau}}{\tau}\right)=\frac{1-\beta^{n+1}}{\tau} \qquad 0\le n\le \tau-1. \qquad (3\text{-}54)$$

$$p_{1,n}=\beta^{n+1}\cdot\left(\frac{1-\beta^{\tau}}{\tau}\right) \qquad n\ge 0. \qquad (3\text{-}55)$$

Since $\sum_{k=0}^{\tau-1}p_{0,k}$ represents the CPU availability for protocol stack processing then this term is expressed as

$$\sum_{k=0}^{\tau-1}p_{0,k}=\frac{\tau-\beta(1-\beta^{\tau})/(1-\beta)}{\tau}. \qquad (3\text{-}56)$$

Similarly, $\sum_{n=0}^{\infty}p_{1,n}$ represents the CPU utilization due to ISR handling which is expressed as

$$\sum_{n=0}^{\infty}p_{1,n}=\sum_{n=0}^{\infty}\beta^{n+1}\left(\frac{1-\beta^{\tau}}{\tau}\right)=\frac{\beta}{\tau}\left(\frac{1-\beta^{\tau}}{1-\beta}\right). \qquad (3\text{-}57)$$

Special case. Let us consider a special case when interrupt handling is generated for each packet, i.e. when $\tau=1$. We prove that equations (3-56) and (3-57) yield the same equations of (3-24) and (3-25).

Substituting $\tau = 1$ in Equation (3-56), then CPU availability for protocol stack processing is

$$\frac{1-\beta(1-\beta^1)/(1-\beta)}{1} = 1-\beta = 1-\frac{\lambda}{\lambda+r} = \frac{r}{\lambda+r}.$$

As for Equation (3-57), CPU utilization due to ISR handling is

$$\frac{\beta}{1}\left(\frac{1-\beta^1}{1-\beta}\right) = \beta = \frac{\lambda}{\lambda+r}.$$

We give some numerical examples of our analytical model to study the impact of interrupt coalescing on CPU usage. For all of these examples, we fix T_{ISR} to 0.3.

We first examine the CPU utilization due to ISR handling with different values of interrupt coalescing parameter τ. In particular when $\tau = 1$, $\tau = 2$, $\tau = 3$, and $\tau = 5$. Notice that $\tau = 1$ means that the system is running in traditional way. Traditional scheme is the one that allows the generation of interrupt for each incoming packet. This scheme was described in section 1.2.2.

Figure 3.32 depicts the impact of interrupt coalescing on CPU utilization. It is noted that as τ increases, the interrupt overhead is decreased.

Figure 3.33 illustrates the relation between CPU availability to process packets in protocol stack and interrupt coalescing for the same system parameter values. It is shown that as τ increase the system has more CPU time to process packets by the kernel protocol stack.

Thus, we conclude that interrupt coalescing reduces interrupt overhead and gives more CPU time for other processes.

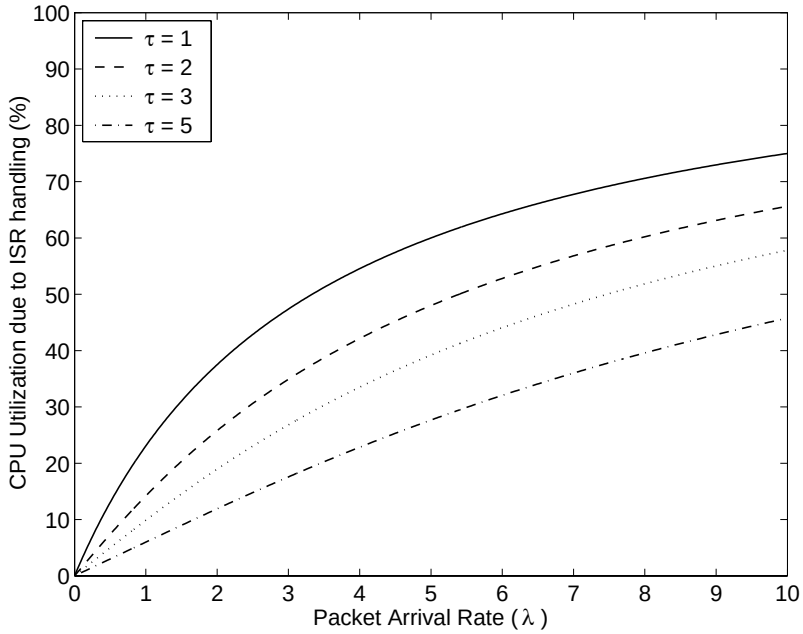


Figure 3.32: CPU utilization due to ISR handling in Interrupt Coalescing scheme

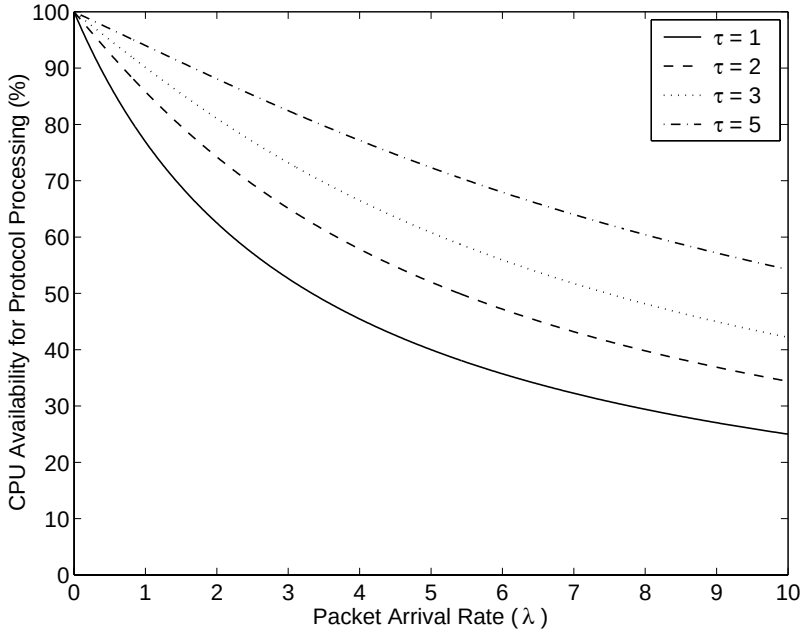


Figure 3.33: CPU availability for protocol processing in Interrupt Coalescing scheme

3.4.2 Modeling Interrupt Coalescing Scheme

The mean effective service rate for interrupt coalescing system (μ'_τ) can be expressed as

$$\mu'_\tau = \mu \cdot \left(\frac{\tau - \beta(1-\beta^\tau)/(1-\beta)}{\tau} \right). \qquad (3\text{-}58)$$

where $\beta = \lambda/(\lambda + r)$.

Since the mean effective service rate is still exponential since the term inside the parenthesis represent fraction of time the CPU is available to process packets in protocol stack layer. Therefore, we can build a Markov chain to model interrupt coalescing scheme with a state space $S = \{(n,m), n \in \{0,1\}, 0 \le m \le \infty\}$. n denotes the server status; either 0 or 1. 0 means that the server is idle waiting for more packets before introducing an interrupt whereas 1 means that the server is processing packets. m denotes the number of packets in the system buffer. Figure 3.34 depicts the rate transition diagram for the Markov chain.

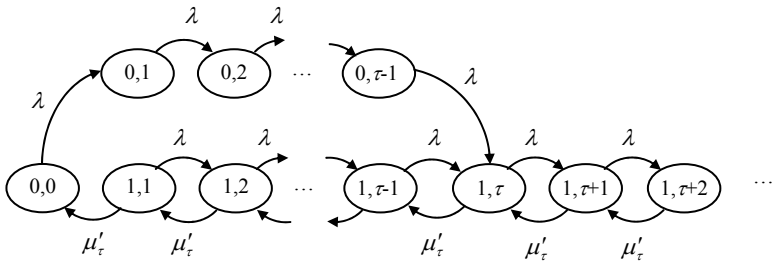


Figure 3.34: States transition diagram for interrupt coalescing scheme

We solve this model by finding the balance equation for each state at a time. At state (0,0), we have

$$-\lambda p_{0,0} + \mu'_\tau p_{1,1} = 0 \qquad \Rightarrow \qquad p_{1,1} = \frac{\lambda}{\mu'_\tau} p_{0,0}.$$

At state (0,1), we have

$$-\lambda p_{0,1} + \lambda p_{0,0} = 0 \qquad \Rightarrow \quad p_{0,1} = p_{0,0} \quad .$$

Similarly with states (0,2), (0,3), …, and (0, τ-1).

$$p_{0,k} = p_{0,0} \qquad 1 \le k \le \tau - 1. \tag{3-59}$$

At state (1,1), we have

$$-(\lambda + \mu'_\tau) p_{1,1} + \mu'_\tau p_{1,2} = 0 \qquad \Rightarrow \qquad \mu'_\tau p_{1,2} = \frac{\lambda^2}{\mu'_\tau} p_{0,0} + \lambda p_{0,0},$$

$$p_{1,2} = \left(\frac{\lambda}{\mu'_\tau}\right)^2 p_{0,0} + \frac{\lambda}{\mu'_\tau} p_{0,0}.$$

At state (1,2), we have

$$-(\lambda + \mu'_\tau) p_{1,2} + \lambda p_{1,1} + \mu'_\tau p_{1,3} = 0,$$

$$\mu'_\tau p_{1,3} = (\lambda + \mu'_\tau)\left[\left(\frac{\lambda}{\mu'_\tau}\right)^2 p_{0,0} + \frac{\lambda}{\mu'_\tau} p_{0,0}\right] - \frac{\lambda^2}{\mu'_\tau} p_{0,0}$$

$$= \frac{\lambda^3}{(\mu'_\tau)^2} p_{0,0} + \frac{\lambda^2}{\mu'_\tau} p_{0,0} + \lambda p_{0,0} \ ,$$

$$p_{1,3} = \left(\frac{\lambda}{\mu'_\tau}\right)^3 p_{0,0} + \left(\frac{\lambda}{\mu'_\tau}\right)^2 p_{0,0} + \frac{\lambda}{\mu'_\tau} p_{0,0}.$$

At state (1,3), we have

$$-(\lambda+\mu'_\tau)\,p_{1,3}+\lambda\,p_{1,2}+\mu'_\tau\,p_{1,4}=0\,,$$

$$\begin{aligned}\mu'_\tau p_{1,4}&=(\lambda+\mu'_\tau)\left[\left(\frac{\lambda}{\mu'_\tau}\right)^3 p_{0,0}+\left(\frac{\lambda}{\mu'_\tau}\right)^2 p_{0,0}+\frac{\lambda}{\mu'_\tau}p_{0,0}\right]-\lambda\left[\left(\frac{\lambda}{\mu'_\tau}\right)^2 p_{0,0}+\frac{\lambda}{\mu'_\tau}p_{0,0}\right]\\&=\frac{\lambda^4}{(\mu'_\tau)^3}p_{0,0}+\frac{\lambda^3}{(\mu'_\tau)^2}p_{0,0}+\frac{\lambda^2}{\mu'_\tau}p_{0,0}+\frac{\lambda^3}{(\mu'_\tau)^2}p_{0,0}+\frac{\lambda^2}{\mu'_\tau}p_{0,0}+\lambda p_{0,0}\\&\quad-\frac{\lambda^3}{(\mu'_\tau)^2}p_{0,0}-\frac{\lambda^2}{\mu'_\tau}p_{0,0}\ ,\end{aligned}$$

$$p_{1,4}=\left(\frac{\lambda}{\mu'_\tau}\right)^4 p_{0,0}+\left(\frac{\lambda}{\mu'_\tau}\right)^3 p_{0,0}+\left(\frac{\lambda}{\mu'_\tau}\right)^2 p_{0,0}+\frac{\lambda}{\mu'_\tau}p_{0,0}\,.$$

Thus, at state (1, n) where $1\le n\le \tau-1$, we have

$$-(\lambda+\mu'_\tau)\,p_{1,n}+\lambda\,p_{1,n-1}+\mu'_\tau\,p_{1,n+1}=0\,,$$

$$\begin{aligned}\mu'_\tau p_{1,n+1}&=(\lambda+\mu'_\tau)\left[\left(\frac{\lambda}{\mu'_\tau}\right)^n p_{0,0}+\left(\frac{\lambda}{\mu'_\tau}\right)^{n-1} p_{0,0}+\cdots+\frac{\lambda}{\mu'_\tau}p_{0,0}\right]\\&\quad-\lambda\left[\left(\frac{\lambda}{\mu'_\tau}\right)^{n-1} p_{0,0}+\left(\frac{\lambda}{\mu'_\tau}\right)^{n-2} p_{0,0}+\cdots+\frac{\lambda}{\mu'_\tau}p_{0,0}\right]\\&=\frac{\lambda^{n+1}}{(\mu'_\tau)^n}p_{0,0}+\frac{\lambda^n}{(\mu'_\tau)^{n-1}}p_{0,0}+\cdots+\frac{\lambda^2}{\mu'_\tau}p_{0,0}\\&\quad+\frac{\lambda^n}{(\mu'_\tau)^{n-1}}p_{0,0}+\frac{\lambda^{n-1}}{(\mu'_\tau)^{n-2}}p_{0,0}+\cdots+\frac{\lambda^2}{\mu'_\tau}p_{0,0}+\lambda p_{0,0}\\&\quad-\frac{\lambda^n}{(\mu'_\tau)^{n-1}}p_{0,0}-\frac{\lambda^{n-1}}{(\mu'_\tau)^{n-2}}p_{0,0}-\cdots-\frac{\lambda^2}{\mu'_\tau}p_{0,0}\ ,\end{aligned}$$

$$p_{1,n+1}=\left(\frac{\lambda}{\mu'_\tau}\right)^{n+1} p_{0,0}+\left(\frac{\lambda}{\mu'_\tau}\right)^n p_{0,0}+\cdots+\frac{\lambda}{\mu'_\tau}p_{0,0}\,. \qquad (3\text{-}60)$$

At state(1, τ), we have

$$-(\lambda+\mu'_\tau)\,p_{1,\tau}+\lambda\,p_{0,\tau-1}+\lambda\,p_{1,\tau-1}+\mu'_\tau\,p_{1,\tau+1}=0\,,$$

$$\mu'_\tau p_{1,\tau+1} = (\lambda+\mu'_\tau)\left[\left(\frac{\lambda}{\mu'_\tau}\right)^{\tau} p_{0,0} + \left(\frac{\lambda}{\mu'_\tau}\right)^{\tau-1} p_{0,0} + \cdots + \frac{\lambda}{\mu'_\tau} p_{0,0}\right]$$
$$- \lambda\left[\left(\frac{\lambda}{\mu'_\tau}\right)^{\tau-1} p_{0,0} + \left(\frac{\lambda}{\mu'_\tau}\right)^{\tau-2} p_{0,0} + \cdots + \frac{\lambda}{\mu'_\tau} p_{0,0}\right] - \lambda\, p_{0,0}$$
$$= \frac{\lambda^{\tau+1}}{(\mu'_\tau)^{\tau}} p_{0,0} + \frac{\lambda^{\tau}}{(\mu'_\tau)^{\tau-1}} p_{0,0} + \cdots + \frac{\lambda^2}{\mu'_\tau} p_{0,0}$$
$$+ \frac{\lambda^{\tau}}{(\mu'_\tau)^{\tau-1}} p_{0,0} + \frac{\lambda^{\tau-1}}{(\mu'_\tau)^{\tau-2}} p_{0,0} + \cdots + \frac{\lambda^2}{\mu'_\tau} p_{0,0} + \lambda p_{0,0}$$
$$- \frac{\lambda^{\tau}}{(\mu'_\tau)^{\tau-1}} p_{0,0} - \frac{\lambda^{\tau-1}}{(\mu'_\tau)^{\tau-2}} p_{0,0} - \cdots - \frac{\lambda^2}{\mu'_\tau} p_{0,0} - \lambda\, p_{0,0} \ ,$$

$$p_{1,\tau+1} = \left(\frac{\lambda}{\mu'_\tau}\right)^{\tau+1} p_{0,0} + \left(\frac{\lambda}{\mu'_\tau}\right)^{\tau} p_{0,0} + \cdots + \left(\frac{\lambda}{\mu'_\tau}\right)^{2} p_{0,0}.$$

Thus

$$p_{1,\tau+n} = \left(\frac{\lambda}{\mu'_\tau}\right)^{\tau+n} p_{0,0} + \left(\frac{\lambda}{\mu'_\tau}\right)^{\tau+n-1} p_{0,0} + \cdots + \left(\frac{\lambda}{\mu'_\tau}\right)^{n+1} p_{0,0} \text{ where } n \geq 0. \qquad (3\text{-}61)$$

Now if we let $\rho_\tau = \lambda / \mu'_\tau$, Equations (3-60) and (3-61) can be simplified as

$$p_{1,n} = p_{0,0} \sum_{i=1}^{n} \rho_\tau^i \text{ where } 1 \leq n \leq \tau - 1.$$

$$p_{1,\tau+n} = \sum_{i=1}^{\tau} \rho_\tau^{n+i} p_{0,0} = \rho_\tau^n p_{0,0} \sum_{i=1}^{\tau} \rho_\tau^i \text{ where } n \geq 0.$$

To find $p_{0,0}$, we use the boundary condition that the summation of all probabilities is equal to 1, i.e.,

$$p_{0,0} + p_{0,1} + \cdots + p_{0,\tau-1} + p_{1,1} + p_{1,2} + p_{1,3} + \cdots = 1,$$

or

$$p_{0,0}\sum_{n=0}^{\tau-1}1+p_{0.0}\sum_{n=1}^{\tau-1}\sum_{i=1}^{n}\rho_\tau^i+p_{0,0}\sum_{k=1}^{\tau}\rho_\tau^k\sum_{n=0}^{\infty}\rho_\tau^n=1.$$

Solving for $p_{0,0}$, we get

$$p_{0,0}=\left[\tau+\sum_{n=1}^{\tau-1}\sum_{i=1}^{n}\rho_\tau^i+\sum_{k=1}^{\tau}\rho_\tau^k\sum_{n=0}^{\infty}\rho_\tau^n\right]^{-1}. \qquad (3\text{-}62)$$

This can be nicely simplified as

$$p_{0,0}=\frac{1-\rho_\tau}{\tau}. \qquad (3\text{-}63)$$

Please notice that the system is idle whenever it is at state (0,0), (0,1), …, (0,τ-1). Thus the idleness of the system can be expressed as

$$p_0=\sum_{n=0}^{\tau-1}\frac{1-\rho_\tau}{\tau}=1-\rho_\tau. \qquad (3\text{-}64)$$

Since we are interested in finding the system throughput for large values of traffic intensity, then we have to model the Markov chain as a finite state space of size B. It is noted that the transition diagram at Figure 3.34 becomes purely M/M/1 after state (1,τ). This implies that the probability at state (1,B) will remain unchanged if we remove state (1,B+1). Therefore, we can bound the geometric series at Equation (3-61) to a finite value B. Hence,

$$p_{0,0}=\left[\tau+\sum_{n=1}^{\tau-1}\sum_{i=1}^{n}\rho_\tau^i+\sum_{k=1}^{\tau}\rho_\tau^k\sum_{n=0}^{B}\rho_\tau^n\right]^{-1}.$$

This can further simplified as

$$p_{0,0} = \frac{(1-\rho_\tau)^2}{\tau + \rho_\tau^{B+\tau+2} - \rho_\tau(\tau + \rho_\tau^{B+1})}. \quad (3\text{-}65)$$

3.4.3 Performance Metrics

1. System throughput. System throughput is expressed as

$$\gamma = \sum_{n=1}^{B} \mu'_\tau \, p_{1,n} = \mu'_\tau \left(1 - \sum_{n=0}^{\tau-1} p_{0,n}\right). \quad (3\text{-}66)$$

2. System latency. The average number of packets in the system can be expressed as

$$E(n) = \sum_n n \, p_{i,n} = \sum_{n=1}^{\tau-1} n \times (p_{0,n} + p_{1,n}) + \sum_{n=0}^{B} (n+\tau) \times p_{1,\tau+n}$$

$$E(n) = \frac{\tau(\tau - \rho(\tau-3) - 1)}{2(\tau + \rho^{B+\tau+2} - \rho(\tau + \rho^{B+1})}. \quad (3\text{-}67)$$

Thus, system latency is expressed as

$$R = \frac{E(n)}{\lambda(1-p_B)}. \quad (3\text{-}68)$$

where p_B is the probability of a packet being dropped due to buffer being full.

3. CPU availability. The CPU availability is the summation of all probabilities when the system is in states (0, *k*) where $0 \le k \le \tau - 1$. Therefore, CPU availability can be expressed as

$$V = \frac{\tau(1-\rho_\tau)^2}{\tau + \rho_\tau^{B+\tau+2} - \rho_\tau(\tau + \rho_\tau^{B+1})}. \qquad (3\text{-}69)$$

4. Overall system power. In order to find overall system power, one has to model the system as infinite state Markov chain. Thus

$$\gamma = \sum_{n=1}^{\infty} \mu'_\tau \, p_{1,n} = \mu'_\tau\left(1 - \sum_{n=0}^{\tau-1} p_{0,n}\right) = \mu'_\tau\left(1 - \frac{1-\rho_\tau}{\tau} \times \tau\right) = \mu'_\tau\left(\frac{\lambda}{\mu'_\tau}\right) = \lambda,$$

$$V(\lambda) = 1 - \rho'_\tau = \frac{\mu'_\tau - \lambda}{\mu'_\tau},$$

$$E(n) = \sum_n n \, p_{i,n} = \sum_{n=1}^{\tau-1} n \times (p_{0,n} + p_{1,n}) + \sum_{n=0}^{\infty} (n+\tau) \times p_{1,\tau+n},$$

$$E(n) = \frac{\tau - (\tau-3)\rho_\tau - 1}{2(1-\rho_\tau)},$$

$$R(\lambda) = \frac{\tau\,\mu'_\tau - (\tau-3)\,\lambda - \mu'_\tau}{2\,\lambda\,(\mu'_\tau - \lambda)}.$$

And therefore

$$P(\lambda) = \frac{2^c\,\lambda^{a+1}\,(\mu'_\tau - \lambda)^{b+c}}{(\mu'_\tau)^b\,(\tau\,\mu'_\tau - (\tau-3)\,\lambda - \mu'_\tau)}. \qquad (3\text{-}70)$$

5. Stability condition: The stability condition is given as

$$\lambda < \mu'_\tau = \mu \cdot \left(1 - \left(\frac{\beta}{1-\beta}\right)\left(\frac{1-\beta^\tau}{\tau}\right)\right),$$

$$\lambda < \mu \cdot \left[1 - \frac{\lambda}{\tau r}\left(1 - \left(\frac{\lambda}{\lambda + r}\right)^\tau\right)\right],$$

$$\lambda - \mu \cdot \left[1 - \frac{\lambda}{\tau r} \left(1 - \left(\frac{\lambda}{\lambda + r} \right)^{\tau} \right) \right] < 0 .$$

This can be rewritten as

$$\lambda(\mu + \tau r)(\lambda + r)^{\tau} - \tau r \mu (\lambda + r)^{\tau} - \mu \lambda^{\tau+1} < 0 . \qquad (3\text{-}71)$$

The left hand side of Equation (3-71) is a polynomial of degree $\tau + 1$. Notice that if we consider $\tau = 1$ the above inequality reduces to Equation (3-30).

3.4.4 Numerical Results

We study in this section the impact of interrupt coalescing on system performance. We compare system performance against Ideal and Traditional schemes. Figure 3.35 depicts system throughput for different values of τ. We observe that as the size of the interrupt coalescing τ increases, the system throughput increases. The amount of increment of maximum system throughput from $\tau = 1$ to $\tau = 2$ is much greater than the amount of increment from $\tau = 2$ to $\tau = 3$. This means increasing τ more than two will not add a significant improvement to the maximum system throughput. We also observe that interrupt coalescing will not prevent livelock but it will shift the livelock point.

Figure 3.36 depicts different behaviors of system latency for different values of τ. At very low arrival rate, the interrupt coalescing has bad behavior in terms of latency. This is because, at lower arrival rate, the time between two successive interrupts is too high. Therefore, packets will remain in the buffer for long time period waiting for

servicing. As packet arrival rate increases the system latency decreases until it reaches its minimum latency. Then, as the arrival rate increases the system increases exponentially.

Figure 3.37 depicts CPU availability for user processes. We notice that as interrupt coalescing size is increased the interrupt overhead is reduced and CPU has more time to process other tasks.

Figure 3.38 shows overall system performance. We observe that as interrupt coalescing size increases the overall system power decreases. In this case, performance degradation is due to high latency. However, at high arrival rate $(\lambda > 0.6)$, interrupt coalescing gives more power than Traditional scheme. Hence, at this rate, it is better to employ interrupt coalescing than using Traditional scheme.

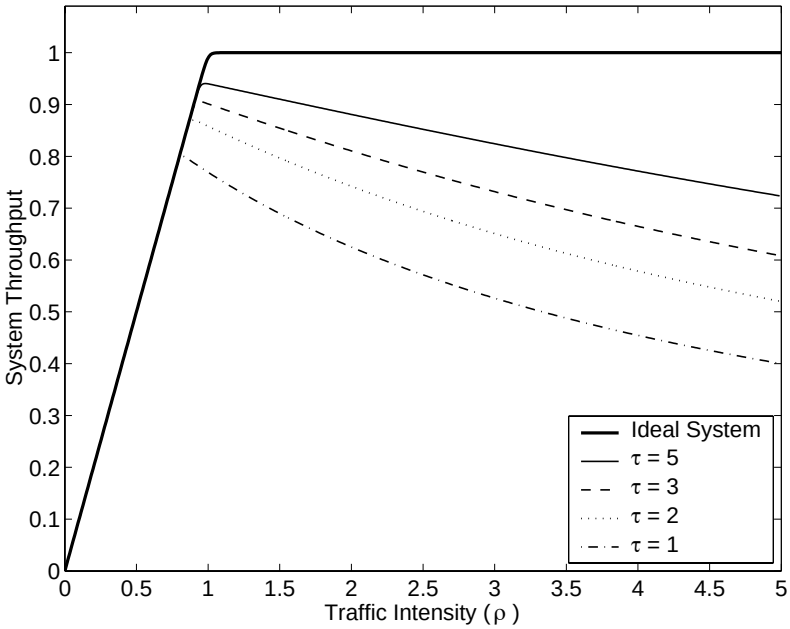


Figure 3.35: System throughput for Interrupt Coalescing scheme

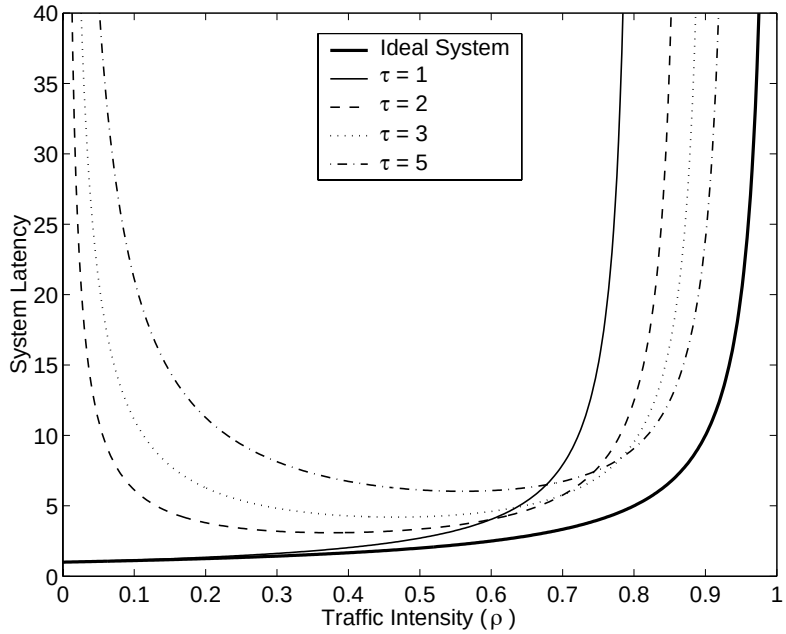


Figure 3.36: System latency for Interrupt Coalescing scheme

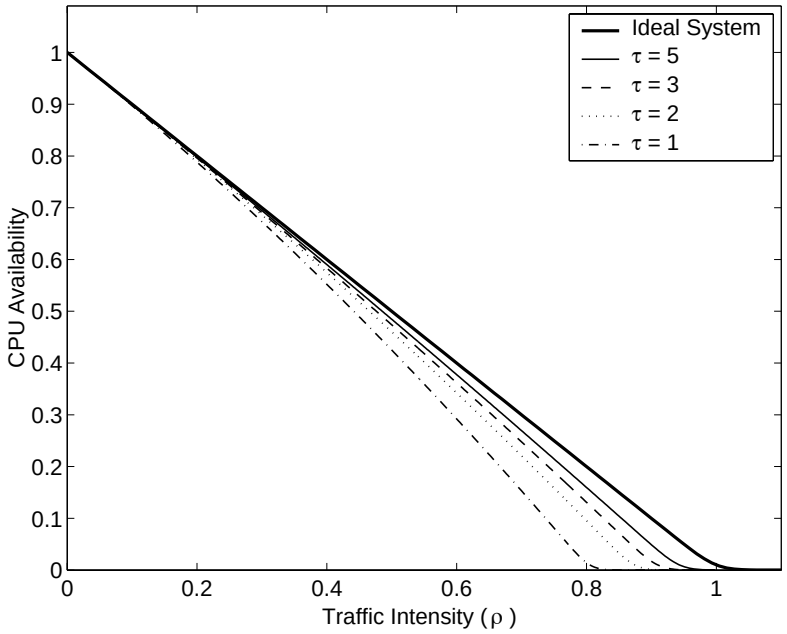


Figure 3.37: CPU availability for Interrupt Coalescing scheme

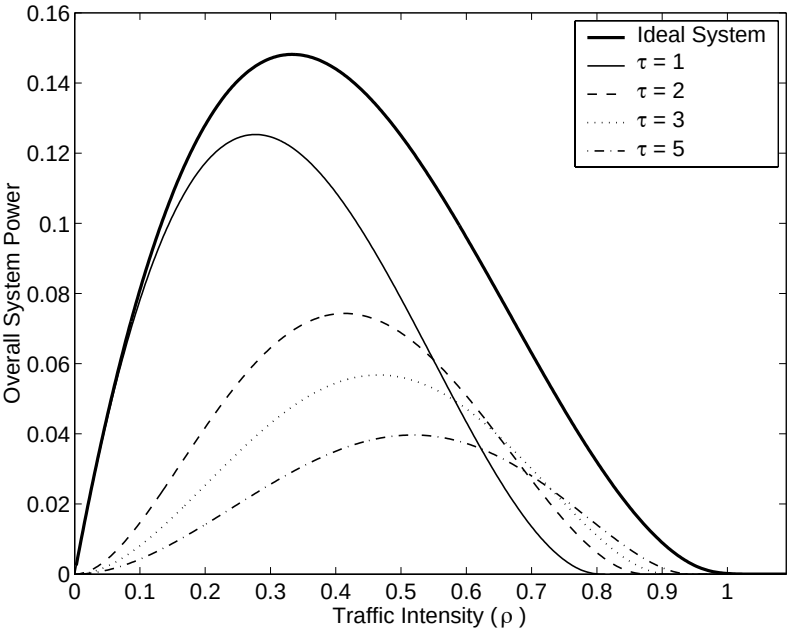


Figure 3.38: Overall system power for Interrupt Coalescing scheme

3.5 Enabling-Disabling Interrupt Model

This section presents an analysis for another proposed solution to mitigate interrupt overhead. The basic idea of this solution relies on disabling interrupts in ISR only and enabling interrupts when system buffer memory becomes empty. Interrupts are disabled when the system memory buffer contains some packets.

Figure 3.39 shows pseudo-code for ISR and packet processing routines for Enabling-Disabling interrupt model. Initially, interrupt status is enabled. When the system receives an incoming packet, the ISR gets executed. ISR disables the interrupt and then it invokes packet processing thread. Packet processing thread starts by processing all packets available in system memory. When the thread finishes packet processing, it will re-enable interrupts again for future incoming packet.

```
ISR() {
      disable_interrupt();
      invoke packet_processing_thread();
      return;
}


packet_processing_thread() {
      while (memory buffer is not empty)
            process_one_packet();
      enable_interrupt();
      return;
}
```

Figure 3.39: Pseudo-code for Enabling-Disabling interrupt scheme

3.5.1 Modeling Enabling-Disabling Interrupt scheme

Let us assume that the time to enable and disable interrupt is T_{INT}. We now consider a model in which it has two mean rates: The first rate applies for servicing only the last packet (i.e. no other packet in buffer), whereas the second rate applies for other packets. The last packet will be served effectively with a time equal to $1/\mu + T_{ISR} + T_{INT}$. The other packets will be served with a time equal to $1/\mu$. For the sake of simplicity, we assume that the service time of the first type is exponentially distributed with mean v. Figure 3.40 illustrates the Markov chain for Enabling-Disabling Interrupt model.

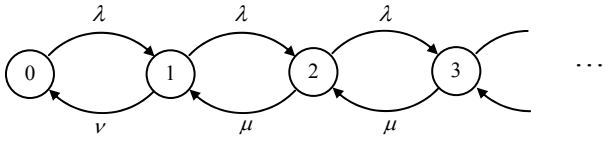


Figure 3.40: Rate-transition diagram for Enabling-Disabling Interrupt scheme

The general solution of this Markov chain is explained in [GRO98], where

$$p_n = \frac{\lambda_1 \lambda_2 \lambda_3 \cdots \lambda_n}{\mu_1 \mu_2 \mu_3 \cdots \mu_n} p_0 .$$

Since $\lambda_i = \lambda$ for all $i > 0$, $\mu_1 = \nu$ and $\mu_i = \mu$ for all $i \geq 2$. Then

$$p_n = \frac{\lambda^n}{\nu \mu^{n-1}} p_0 = \left(\frac{\lambda}{\nu}\right)\left(\frac{\lambda}{\mu}\right)^{n-1} p_0 \qquad (3\text{-}72)$$

Let $\rho_1 = \lambda / \nu$ and $\rho_2 = \lambda / \mu$, then

$$p_n = \rho_1 \rho_2^{n-1} p_0 \qquad (3\text{-}73)$$

For obtaining p_0, we use the boundary condition,

$$p_0 + \sum_{n=1}^{\infty} \rho_1 \rho_2^{n-1} p_0 = 1 .$$

Solving for p_0, we have

$$p_0 = \left[1 + \rho_1 \sum_{n=1}^{\infty} \rho_2^{n-1} \right]^{-1} = \left[1 + \frac{\rho_1}{1-\rho_2} \right]^{-1} .$$

Thus,

$$p_0 = \frac{1-\rho_2}{1+\rho_1-\rho_2} . \qquad (3\text{-}74)$$

Note that, p_0 is valid only when $\rho_2 < 1$. For the case of high traffic intensity when $\rho_2 > 1$, we have to model it as M/M/1/B. Thus, p_0 is modified as

$$p_0 = \left[1 + \rho_1 \sum_{n=1}^{B} \rho_2^{n-1} \right]^{-1} = \left[1 + \frac{\rho_1 (1-\rho_2^B)}{1-\rho_2} \right]^{-1} ,$$

hence

$$p_0 = \frac{1-\rho_2}{1+\rho_1-\rho_2-\rho_1\rho_2^B} . \qquad (3\text{-}75)$$

3.5.2 Performance Metrics

1. System throughput. By applying Equation (3-2), system throughput can be expressed as

$$\gamma = v\rho_1 p_0 + \mu\rho_1 p_0 \sum_{n=2}^{B} \rho_2^{n-1} = \rho_1 p_0 \cdot \left[\frac{v(1-\rho_2) + \lambda(1-\rho_2^{B-1})}{1-\rho_2} \right],$$

which can be simplified to

$$\gamma = \lambda \cdot \left[\frac{1+\rho_1-\rho_2-\rho_1\rho_2^{B-1}}{1+\rho_1-\rho_2-\rho_1\rho_2^{B}} \right]. \qquad (3\text{-}76)$$

2. System latency. The average number of packets in the system can be obtain as follows:

$$E(n) = \sum_{n=1}^{B} np_n = \rho_1 p_0 \sum_{n=1}^{B} n\rho_2^{n-1} = \frac{\rho_1 - (B+1)\rho_1\rho_2^{B} + B\rho_1\rho_2^{B+1}}{(1-\rho_2)(1+\rho_1-\rho_2-\rho_1\rho_2^{B})}.$$

Thus, the system latency is followed by using Equation (3-10).

3. CPU availability. CPU availability is expressed as

$$V = \frac{1-\rho_2}{1+\rho_1-\rho_2-\rho_1\rho_2^{B}}. \qquad (3\text{-}77)$$

4. Overall system power. We want to express system throughput, CPU availability, and system latency as infinite state space system. Thus

$$\begin{aligned} \gamma(\lambda) &= v\rho_1 p_0 + \mu\rho_1 p_0 \sum_{n=2}^{\infty} \rho_2^{n-1} \\ &= \rho_1 p_0 \cdot \left(\frac{v - v\rho_2 + \mu\rho_2}{1-\rho_2} \right) \\ &= \frac{\lambda - \lambda\rho_2 + \lambda\rho_1}{1+\rho_1-\rho_2} \\ &= \lambda , \end{aligned}$$

$$V(\lambda) = \frac{1-\rho_2}{1+\rho_1-\rho_2},$$

and

$$R(\lambda) = \frac{\rho_1}{(1-\rho_2)(1+\rho_1-\rho_2)\cdot\lambda}.$$

Then, the overall system power is expressed as

$$P(\lambda) = \frac{\lambda^{a+c}(1-\rho_2)^{b+c}(1+\rho_1-\rho_2)^{c-b}}{\rho_1^c}. \qquad (3\text{-}78)$$

To obtain the maximum power point with all the tunable parameters are equal to 1, we get

$$P(\lambda) = \frac{\lambda^2\cdot(1-\rho_2)^2}{\rho_1} = v\lambda\cdot\left(1-\frac{\lambda}{\mu}\right)^2.$$

Taking derivative of $P(\lambda)$,

$$\frac{dP}{d\lambda} = v\left(1-\frac{\lambda}{\mu}\right)^2 - 2v\frac{\lambda}{\mu}\left(1-\frac{\lambda}{\mu}\right).$$

Putting $dP/d\lambda = 0$, we have

$$v\left(1-\frac{\lambda}{\mu}\right)\cdot\left(1-3\frac{\lambda}{\mu}\right) = 0$$

Thus, the optimal operation point occurs when $\rho = 1/3$. This point is independent of interrupt overhead. Also, this point is exactly the same operating point for Ideal system.

5. Stability condition. This system will stable whenever $\lambda < \mu$.

3.5.3 Numerical Results

In this section, we show some numerical results of our analytical model to study the system performance of Enabling-Disabling Interrupt. In all of these results, we fix T_{INT} to 0.05, μ to 1, and B to 100.

Figure 3.41 depicts the system throughput as a function of traffic intensity ρ. We study this relation for three T_{ISR} time units 0.2, 0.3, and 0.5. We note that the throughput is not affected by interrupt overhead. The system throughput behaves exactly as the Ideal system. Notice that Figure 3.41 shows only one graph for system throughput because other graphs are hidden behind this graph.

Figure 3.42 shows the CPU availability for Enabling-Disabling Interrupt scheme. As shown, the CPU availability diminished at $\rho = 1$, in spite of the interrupt overhead. But, we observe that CPU availability starts decreasing because, at low rate, packets are processed before a new packet comes to the system. In this situation, the system will introduce an extra overhead due to enabling and disabling interrupts. When packet arrival rate increases such that the buffer keeps nonempty, the ISR overhead, enabling interrupt overhead and disabling interrupt overhead are eliminated.

Figure 3.43 shows the relation between system latency and traffic intensity for the same system parameter values considered for system throughput. It is shown that the latency for Enabling-Disabling interrupt system is very close to the Ideal system latency.

Figure 3.44 illustrates the relation between the overall system power and traffic intensity. Note that the overall system power decreases as interrupt overhead increases. Note also that the maximum power point occurs at $\rho = 1/3$ despite of T_{ISR} value. This point matches exactly the maximum power point we derive it mathematically.

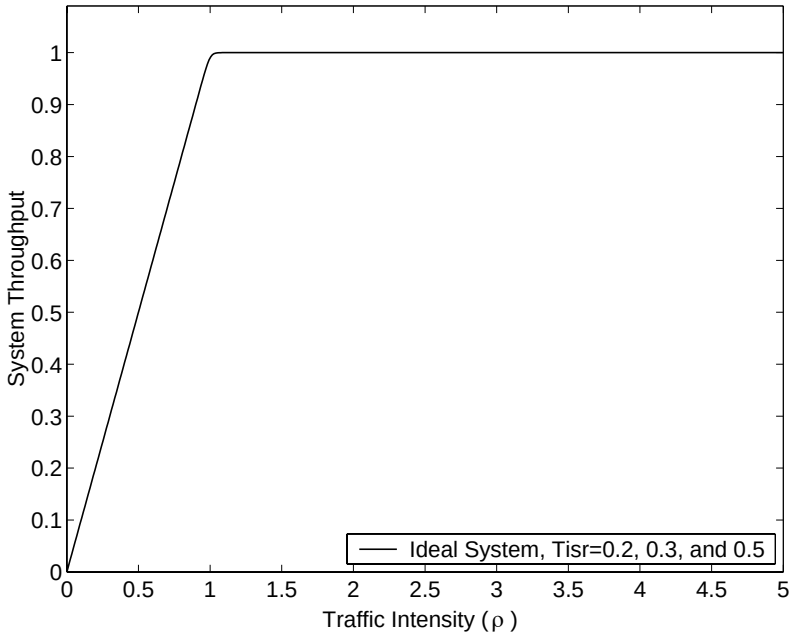


Figure 3.41: System throughputs for Enabling-Disabling Interrupt scheme

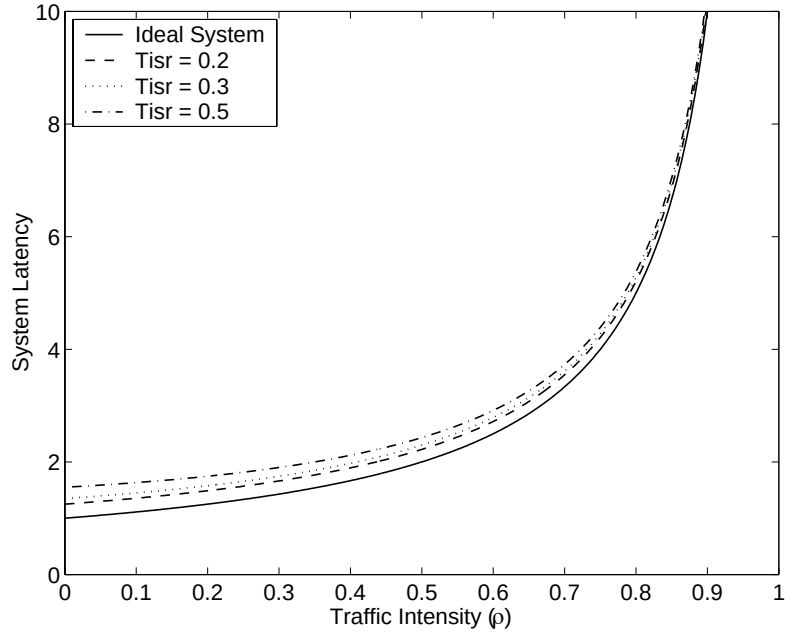


Figure 3.42: System latency for Enabling-Disabling Interrupt scheme

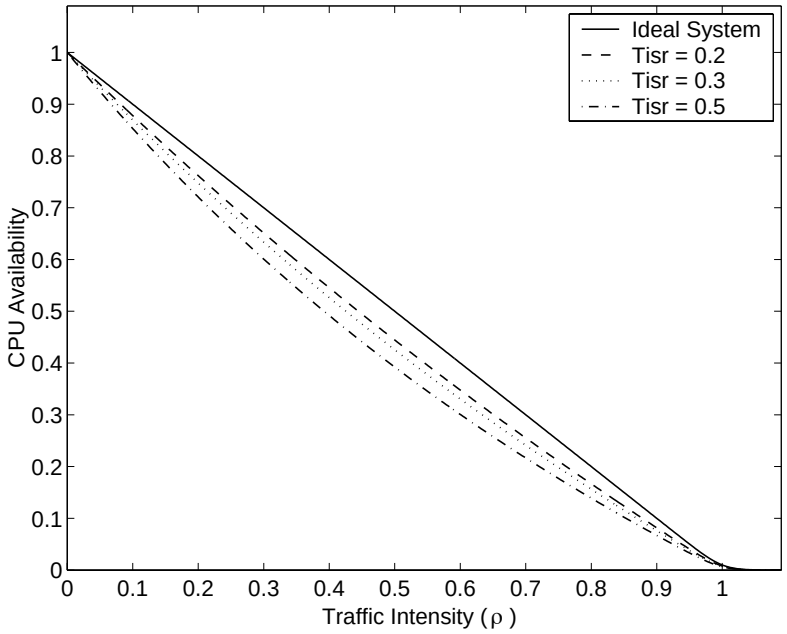


Figure 3.43: CPU availability for Enabling-Disabling Interrupt scheme

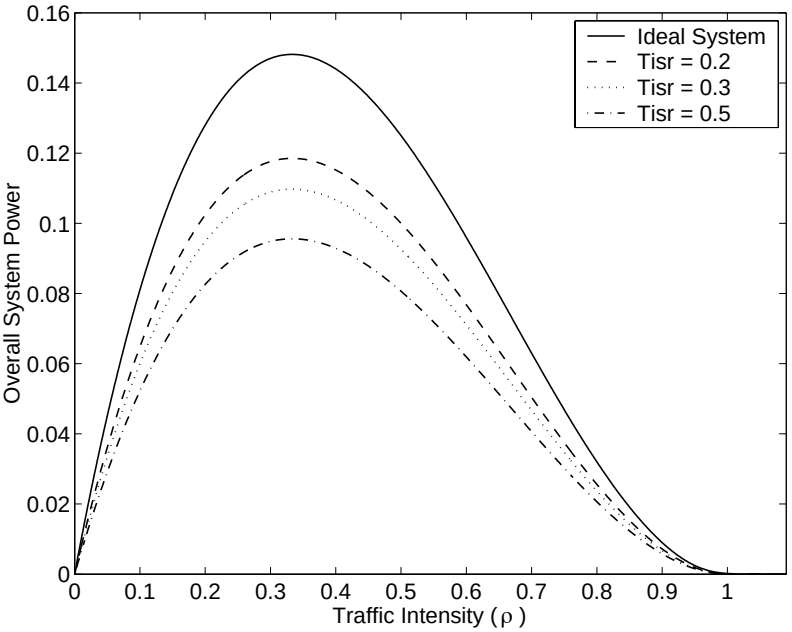


Figure 3.44: Overall system power for Enabling-Disabling Interrupt scheme

CHAPTER 4

SIMULATION STUDY

This chapter covers simulation. Topics include the simulation type, implementation language, selection of random number generators and seeds. The simulation model's components, organization, and logic are presented. Comparison of analytical models and simulation results are also presented. The source code for the implementation of the simulation is found in Appendix B.

4.1 Introduction

This section gives background information on a few necessary topics used for implementation of our simulation. These topics include simulation type, simulation language, random number generators, random-number streams and seed selection.

4.1.1 Simulation Type

Our simulation is a discrete-event simulation, i.e., it uses a discrete-event simulation model. This is opposite to a continuous-event simulation in which the state of the system takes continuous values. Our simulation is a discrete-event simulation since the state of the system is described by variables that do not take continuous values. The state variables change instantaneously at separate points in time. Some of these

variables include number of packets in a system, CPU state herein refer to as server state (idle or busy) and so on. Details of these variables and their use will be described later in this chapter.

4.1.2 Simulation Language

A number of simulation languages were considered for implementation. These languages include OPNET, SLAM II, MATLAB and C. Simulation languages such as OPNET do not offer flexibility and require considerable time in learning the language. SLAM II did not offer flexibility either and was limited in options to perform all aspects of our simulation. A general-purpose language such as MATLAB was a good candidate. It has a powerful, comprehensive and easy-to-use environment for performing technical computations. It has plotting capabilities which are necessary features that would make simulation valid very easy. For these reasons, MATLAB was used initially, however implementation and debugging got a bit tedious and cumbersome when it came to handling queues. For this reason, the C language was chosen. C provides the best flexibility in coding the simulation. However, a considerable amount of effort had to be put into queue implementation, random number generation, and other features necessary for simulation validation.

4.1.3 Random Number Generator

The Inverse Transformation [JAI91] method was used to generate random number varieties for our probability distribution (i.e. exponential distribution). The

Inverse Transformation method uses uniform deviates, U(0,1). Uniform deviates are random numbers that are uniformly distributed between 0 and 1. The Inverse Transformation for Exponential distribution with CDF $F(x) = 1 - e^{-x/a}$ is expressed as:

$$-a\ln(U) \qquad (4\text{-}1)$$

Hence, a reliable source of random uniform deviates is an essential building block for our simulation. Although, the ANSI C library provides a random generation function, *rand*( ), which can be used for generating random deviates, it is quite flawed and totally botched, according to [PRE94]. The authors in [LAW91] recommended the use of PMMLCG (*prime modulus multiplicative linear congruential generator*). The basic algorithm is described in [PAR88]:

$$seed = (7^5\, seed) \bmod (2^{31} - 1) \qquad (4\text{-}2)$$

The PMMLCG is a more efficient generator than the LCG. The LCG is one of the most popular methods for generating random numbers, according to [LAW91]. Also MATLAB uses this generator. Hence, PMMLCG is used for our simulation. The C source code for implementing the PMMLCG is included in the lcg.c module in Appendix B .

4.1.4 Seed Selection

Proper seed selections have to be made in order to avoid wrong combinations of seeds and random number generators that may lead to erroneous results. Care was taken in selecting seeds for multiple random-number streams. A different stream is generated

for each simulation variable. Here are briefly some of the guidelines that are followed in selecting seeds, see [JAI91]:

- Arbitrary values for seeds were not used. Also, the values of zero and even values were not used.
- Every simulation variable has its own stream, and streams were not subdivided.
- Overlapping of streams, to prevent correlation, was avoided by choosing seeds spaced 100,000 apart. In our case, the seeds were spaced 800,000 apart.
- Each simulation iteration did not have to reinitialize seeds. Leftover seeds from previous iterations were used.

These guidelines were followed in implementing the simulation.

4.2 Components and Organization

In this section, we develop a discrete-event simulation model for interrupt-driven kernel. Simulation models for Traditional, Interrupt Coalescing, and Enabling-Disabling Interrupt schemes are developed. Figure 4.1 depicts the general flowchart of the simulation model that is applied for all interrupt handling schemes. Before diving into the details of simulation logic, we would like to discuss the main components used in our discrete-event simulation model:

1. *System state*: Several variables used to describe the current state of the system. Two variables are used to describe the current status of the server;

isr_handling_status and *protocol_processing_status*. The first state describes whether the server is handling ISR or not. The second state describes whether the server is processing a packet in kernel protocol stack or not. These two states are either set to busy or idle. If the server is busy handling ISR, this means *isr_handling_status* is set to busy. If the server is busy processing a packet, this means *protocol_processing_status* is set to busy. Please note the two variables may be in busy. This means the server is handling an ISR and the existing packet processing in kernel protocol stack has already been preempted by ISR.

2. *Events*: Our simulation model has three types of events, shown in Figure 4.2. ARRIVAL event occurs when a new packet arrives to the system. ISR event occurs when the server returns from ISR. DEPARTURE event occurs when a packet is completely processed by the kernel protocol stack. All these events are generated independently. This means that each event has its own seed and random-number stream.

3. *Statistical variables*: Several statistical results of system performance are needed to be gathered from simulation model. These include server utilization due to protocol stack processing, server utilization due ISR handling, average number of packets in the system, total response time, and total number of packets departs the system.

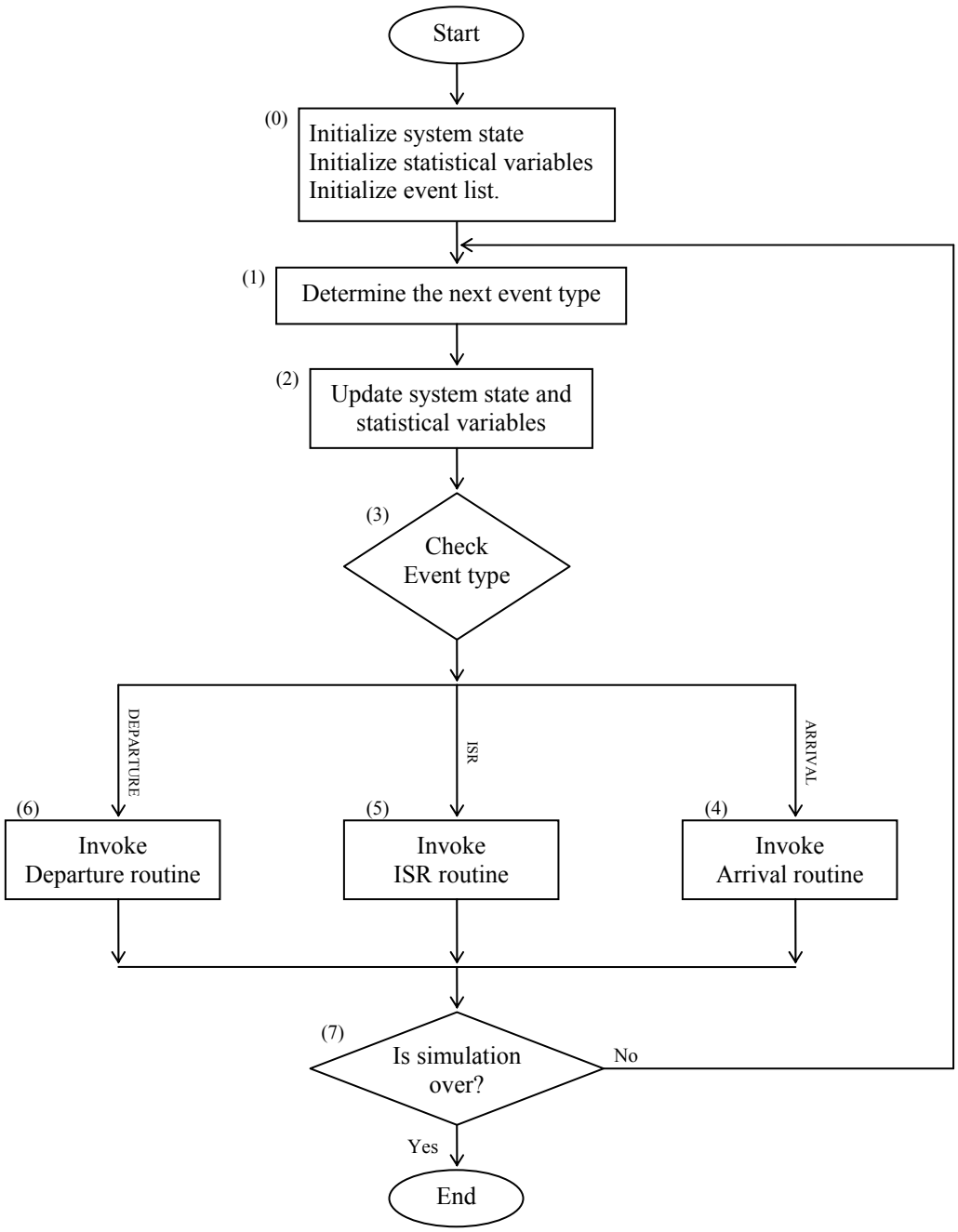


Figure 4.1: Flowchart of the Simulation Model

```
#define ARRIVAL     1  /* packet arrival event */
#define ISR         2  /* ISR handling has been finished */
#define DEPARTURE   3  /* packet departure event */
```

Figure 4.2: C Declaration for event types

4. *Queues*: The simulation has two types of queues: priority and FIFO. A priority queue is used to process next events. The priority, which is based on a "time" value and system status, follows the following criteria. If the system state is currently handling an ISR, then DEPARTURE event cannot be scheduled next. Only ARRIVAL or ISR will be scheduled next based on the most imminent of these two events. If the system status is not handling an ISR, then either ARRIVAL or DEPARTURE event will be scheduled based on the most imminent of these two events. On the other hand, FIFO queue is used to store the times of arrival of packets currently in the systerm. These times are used for statistic gathering.

Next we will discuss the simulation logic in general that is valid for any interrupt-driven system to be modeled. We use a next-event time advance for incrementing our simulation clock. This means the simulation clock is initialed to zero and the times of occurrence of future events are determined. The simulation clock is then advanced to next event according the criteria we mentioned in previous paragraph.

The details of the simulation are given in Figure 4.1. The simulation starts initializing all system components (step 0). The next event is determined and the simulation clock is advanced to the time of the selected event. Consequently, the statistical variables are updated (step 2). Then, the type of the next event is checked and the appropriate event handler is invoked. The handling of these events depends on the

employed scheme to be modeled. Finally, the simulation process from step 1 up to step 6 will be repeated 800,000 times.

4.3 Traditional Scheme Simulation Model

We first simulate the Traditional scheme. Figure 4.3 depicts the flowcharts of simulation model for Traditional scheme.

The simulation logic for this model is as follows. When ARRIVAL event is triggered, we first schedule the next ARRIVAL event. Then, the number of packet arrivals is incremented by one (step 8). Next, the server state is checked in step 9. If the server is busy handling ISR, i.e., the packet arrival occurs during ISR handling, then generating an interrupt is ignored for this packet. Otherwise, we set the ISR handling status to busy and we schedule the finishing time for this ISR (step 10). Finally, if the FIFO queue is not full, we insert the arrival time of this packet in the FIFO queue. The reason of this storing is to keep track the arrival time for each packet in order to compute packet delay. The packet delay is determined by subtraction of arrival time from departure time.

When ISR event is triggered, we first reset the ISR handling status to idle (step 13), i.e., the server is finished handling the ISR. Then, protocol processing status is checked in step 14. The reason of this checking is to see if this ISR preempts packet processing in protocol stack or not. If the status of the protocol processing is busy, i.e., the server was busy processing a packet in protocol stack before interrupt disruption, then the departure time of the preempted packet will be delayed by ISR time (step 16).

If the status of protocol processing is idle, i.e., no packet has been interrupted during its processing in protocol stack, then we schedule the DEPARTURE event for this packet and we change the protocol processing status to busy.

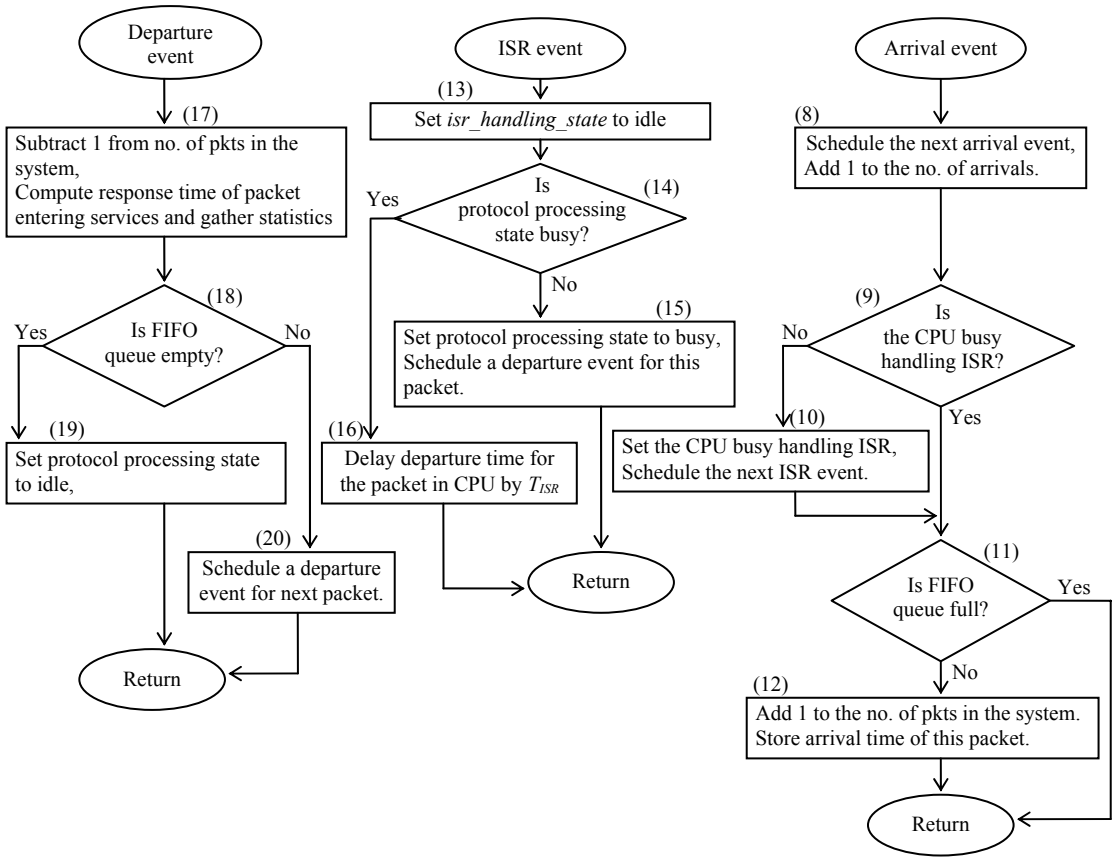


Figure 4.3: Flowcharts of event handlers in Traditional scheme

The last event is DEPARTURE event. When this event is scheduled next, we reduce the number of packets in the system by one and update some statistical information (step 17). Then, we check if FIFO queue is empty or not (step 18). If FIFO queue is not empty, we schedule the departure time of the next packet in the queue (step 20). The next packet is the packet that its arrival time is stored at the top of the queue.

If FIFO queue is empty, we set the server protocol stack status to idle and eliminate the departure event from priority queue (step 19).

4.4 Interrupt Coalescing Simulation Model

To simulate this model, two new components have been added to system states: *coalescing size* and *coalescing counter*. *Coalescing size* represents how many packets to be received before generating an interrupt. *Coalescing counter* counts the number of packets have been received so far. Whenever *coalescing counter* reaches the value indicated in *coalescing size*, the system will generate an interrupt. These two states are initialized at step 0 in the main flowchart, shown in Figure 4.1 in which the *coalescing size* is initialized to the predefined number of packets to be coalesced before generating an interrupt and *coalescing counter* is initialized to zero.

Since Interrupt Coalescing scheme does not affect ISR and DEPARTURE event handlers, the only change we need to modify from Traditional scheme is the ARRIVAL event handler. Figure 4.4 shows the modified version of ARRIVAL event handler for simulating Interrupt Coalescing model.

The logical steps of Interrupt Coalescing are as follows. First, we schedule the event of next packet arrival. Then, we increment the number of packet arrivals and *coalescing counter* by one (step 8). Next, we check the value of *coalescing counter* (step 9). If *coalescing counter* equals to *coalescing size*, then we reset *coalescing counter* to zero (step 10). Then, we generate an interrupt by apply the same steps

mentioned in section 4.3. If *coalescing counter* is not equal to *coalescing size*, then we just insert the arrival of time of this packet into the FIFO queue.

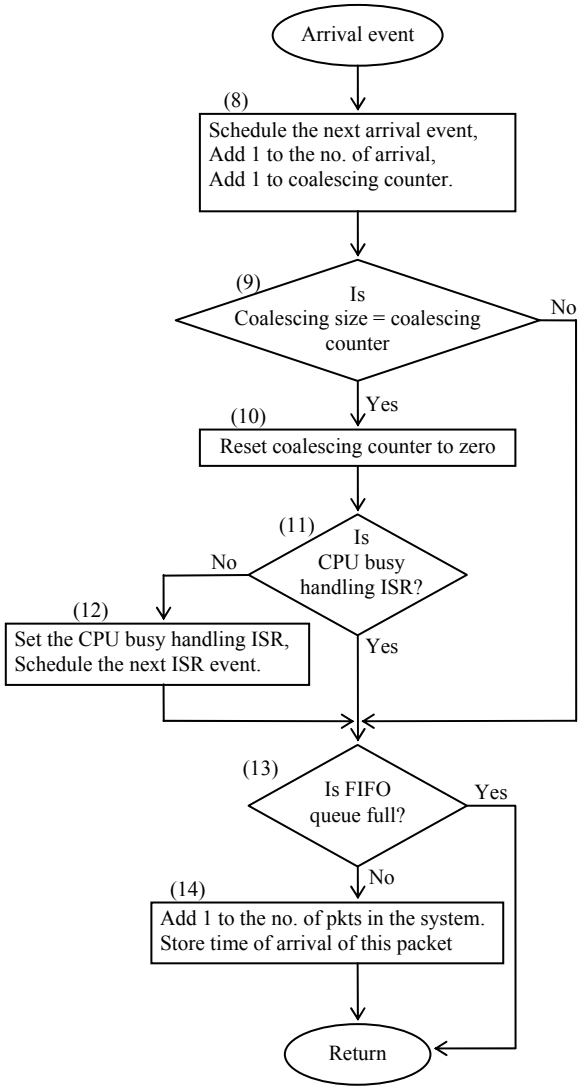


Figure 4.4: Flowchart of ARRIVAL event for Interrupt Coalescing model

4.5 Enabling-Disabling Interrupt Scheme Simulation Model

We now consider a simulation model for Enabling-Disabling Interrupt. In this model, a new variable is needed to indicate interrupt status: enabled or disabled. Initially, the interrupt status is enabled. This means an incoming packet will generate an interrupt. The interrupt status is initialized in the main flowchart at step 0 as shown in Figure 4.1. Figure 4.5 shows the flowcharts of Enabling-Disabling Interrupt scheme simulation model for ARRIVAL and DEPARTURE events. The ISR flowchart is the same as in Figure 4.3.

The logical steps of Enabling-Disabling Interrupt scheme are as follows. Whenever an arrival event is triggered, the system checks the interrupt status whether it is enabled or not (step 9). If the interrupt status is disabled then the arrival time of packet is directly inserted in the FIFO queue (step 12). If the interrupt status is enabled, then it will be changed to disabled and an interrupt will be generated (step 10).

When DEPARTURE event is scheduled next, we reduce the number of packets in the system by one and update some statistical information (step 17). Then, we check if FIFO queue is empty or not (step 18). If FIFO queue is not empty, we schedule the departure time of the next packet in the queue (step 20). If FIFO queue is empty, we set the server protocol stack status to idle and we enable the interrupt again (step 19).

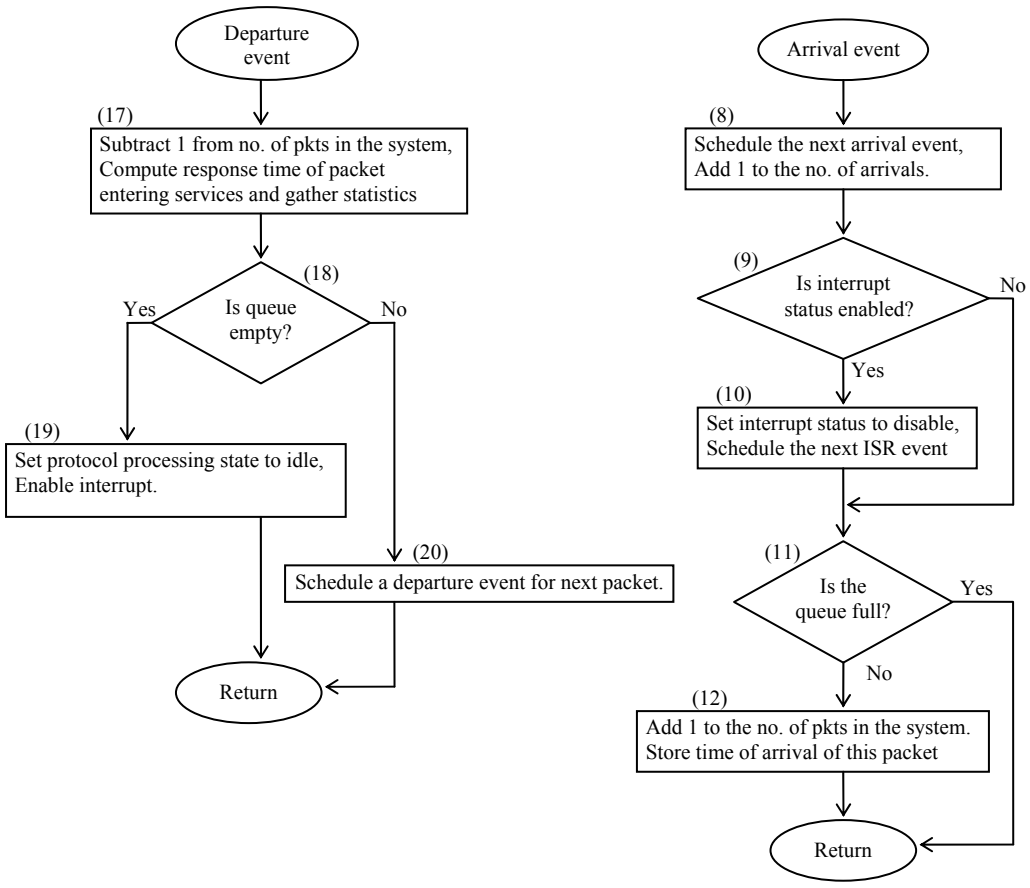


Figure 4.5: Flowcharts of ARRIVAL and DEPARTURE events for Enabling-Disabling Interrupt model

4.6 Comparison and Numerical Results

We now compare numerical results obtained by both analysis and simulation for studying the performance of interrupt-driven kernel. We ran a simulation for a long time period until it generated 800,000 events. In all our results, we fixed T_{ISR} to 0.3 and B = 100. We plot the simulation results with the equivalent figure presented in Chapter 3.

Figure 4.6, Figure 4.7, and Figure 4.8 depict the comparison between analysis and simulation for the first scheme used to model Traditional scheme. It is shown that the analysis and simulation are identical for system throughput. For CPU availability and latency, simulation results are very close to analytical results.

Figure 4.9, Figure 4.10, and Figure 4.11 show the comparison between analysis and simulation for the second scheme used to model Traditional scheme. It is noted that the results given by second analytical model match precisely the results given by simulation.

Figure 4.12, Figure 4.13, and Figure 4.14 depict the comparison between analysis and simulation for Interrupt Coalescing scheme. It is shown that the two models are quit similar especially for system throughput and system latency.

Figure 4.15, Figure 4.16, and Figure 4.17 show the comparison between analysis and simulation for Enabling-Disabling Interrupt scheme. It is noted that the results given by analytic model match the results given by simulation.

We conclude that a perfect accordance has been verified between analysis and simulation. The results given by simulation match precisely the same ones given by derived equations for system throughput and system latency.

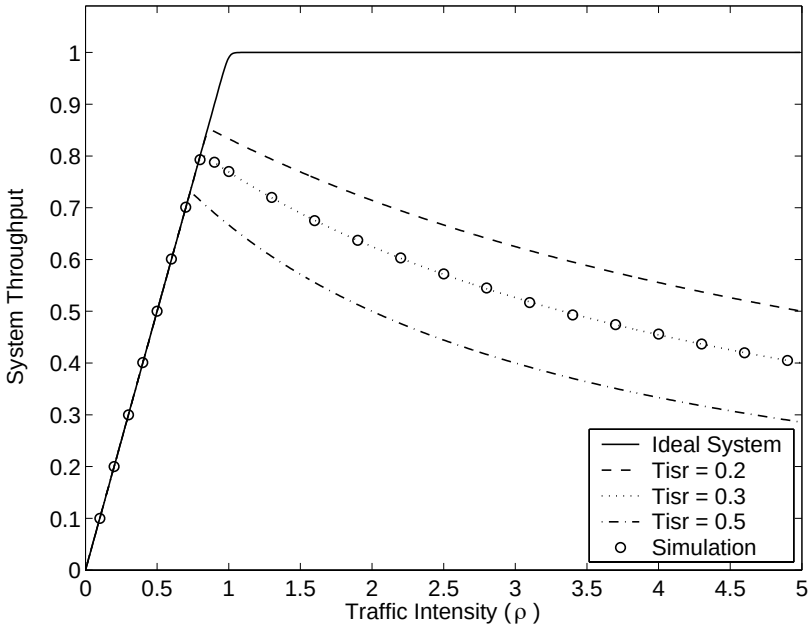


Figure 4.6: Comparison between analysis and simulation of the first Traditional system model for system throughput

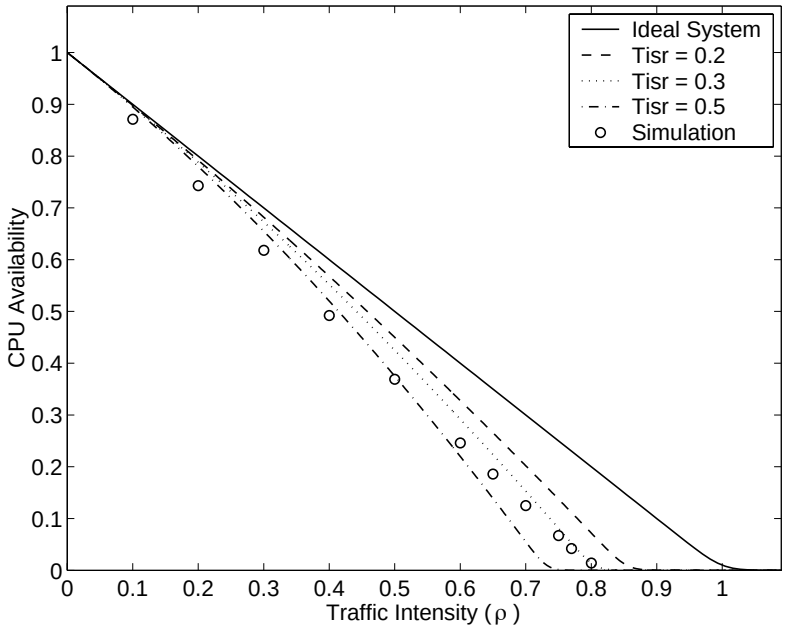


Figure 4.7: Comparison between analysis and simulation of the first Traditional system model for CPU availability

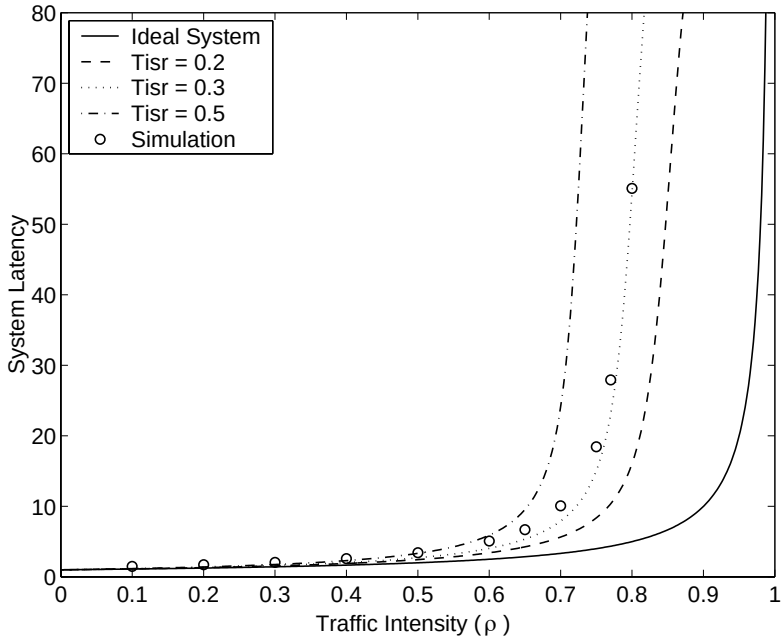


Figure 4.8: Comparison between analysis and simulation of the first Traditional system model for system latency

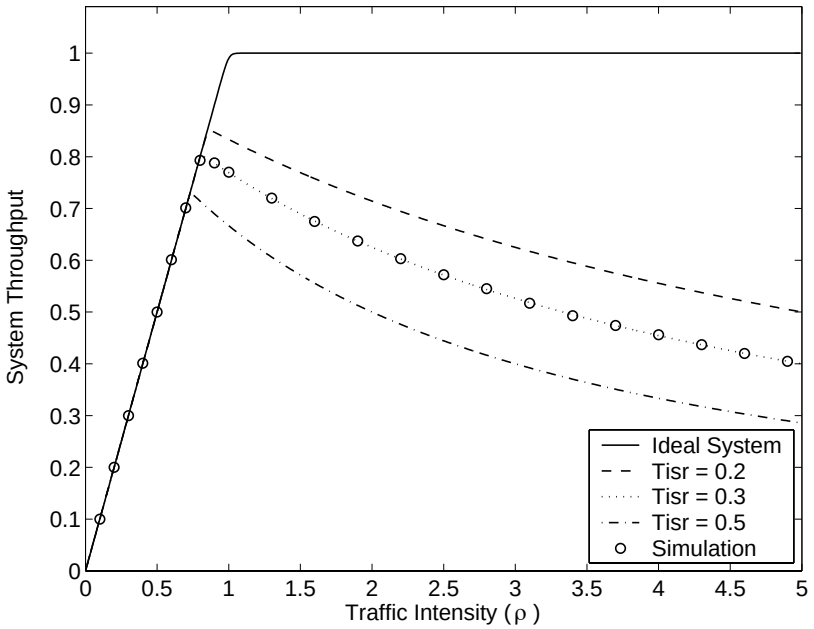


Figure 4.9: Comparison between analysis and simulation of the second Traditional system model (first solution) for system throughput

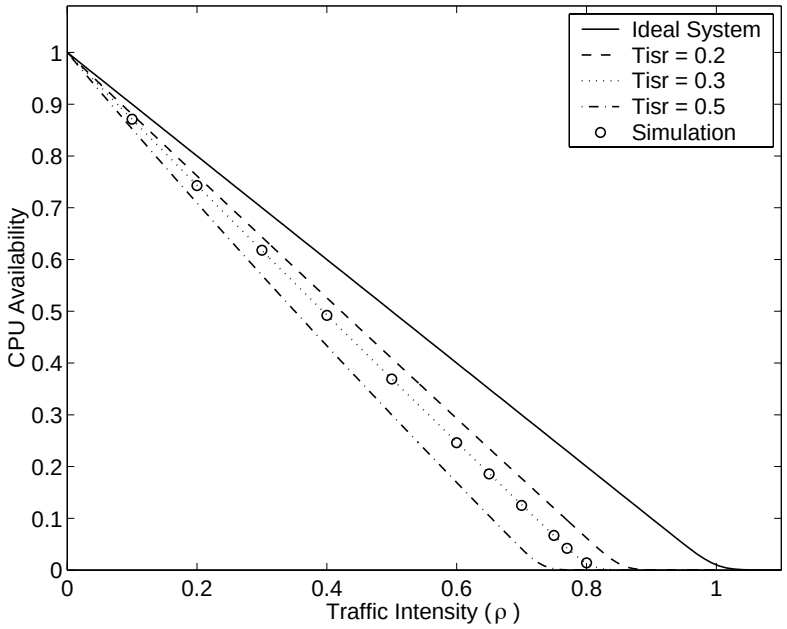


Figure 4.10: Comparison between analysis and simulation of the second Traditional system model for CPU availability

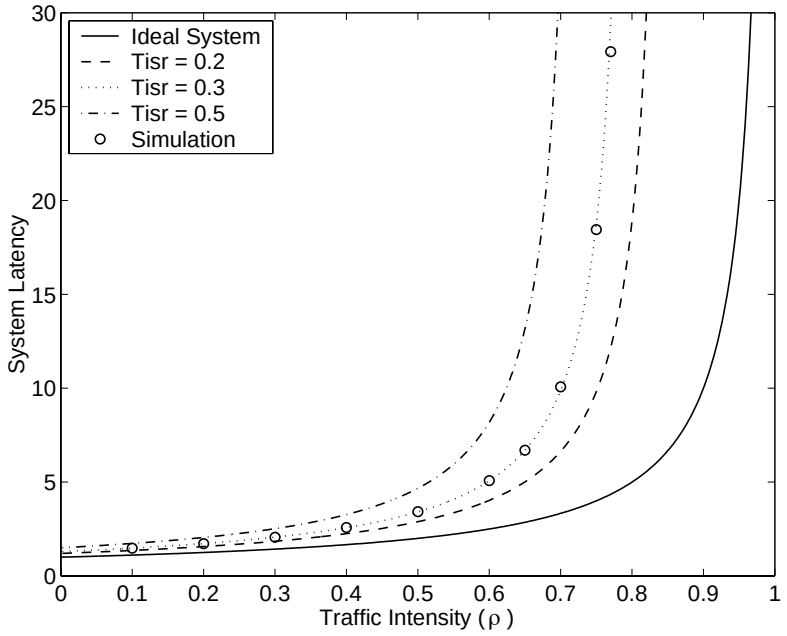


Figure 4.11: Comparison between analysis and simulation of the second Traditional system model for system latency

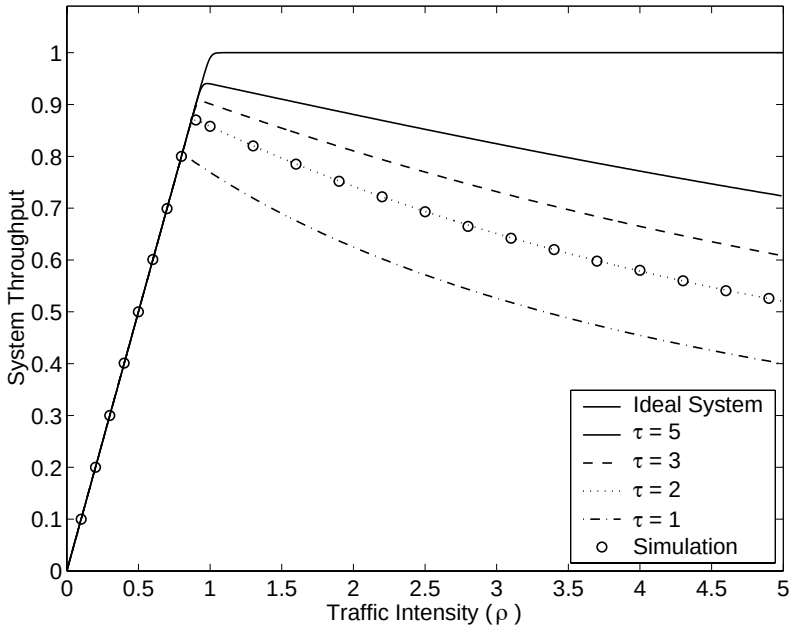


Figure 4.12: Comparison between analysis and simulation of Interrupt Coalescing model for system throughput

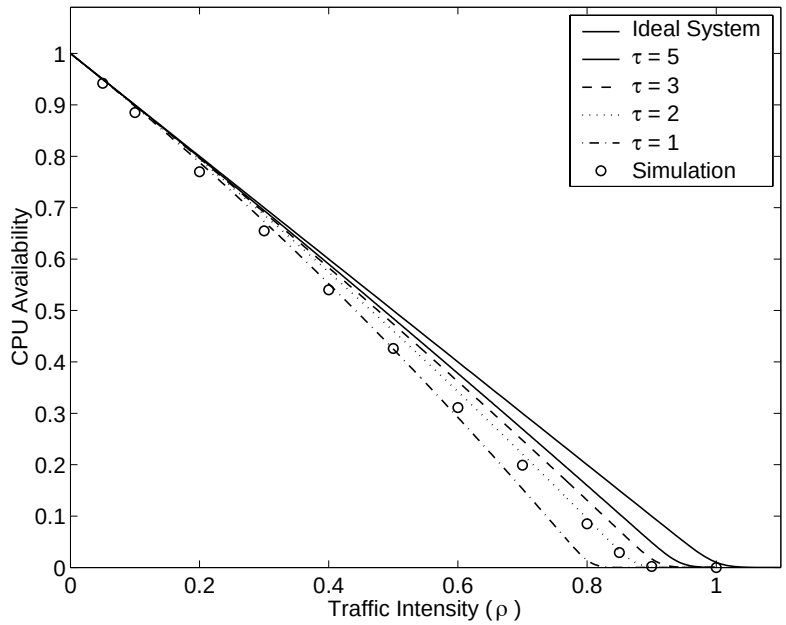


Figure 4.13: Comparison between analysis and simulation of Interrupt Coalescing model for CPU availability

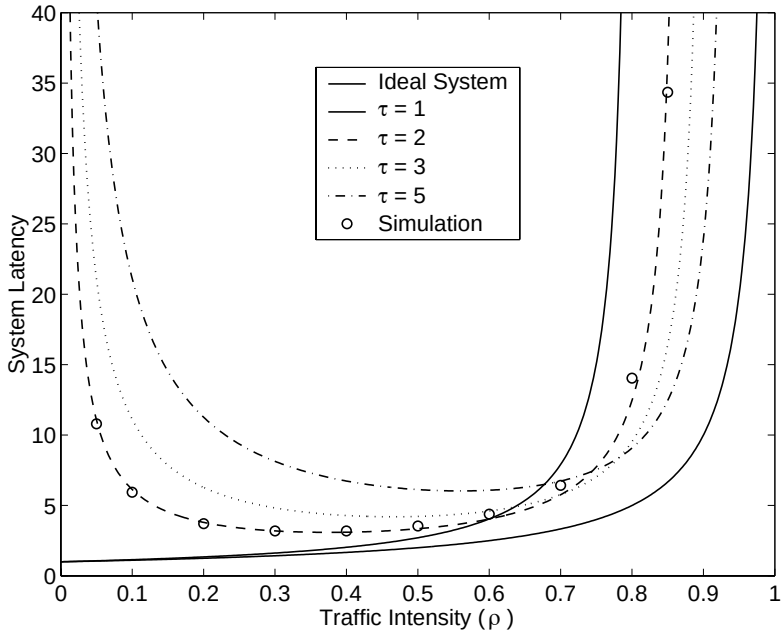


Figure 4.14: Comparison between analysis and simulation of Interrupt Coalescing model for system latency

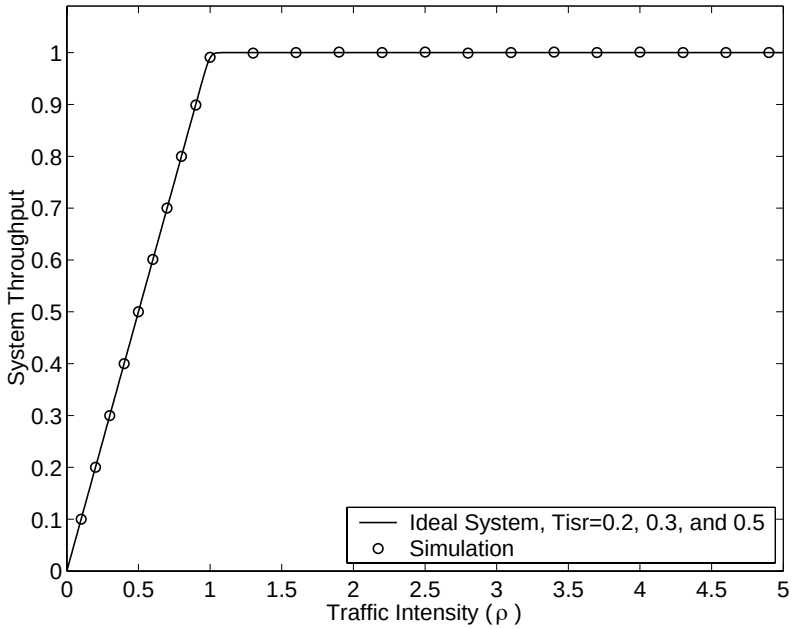


Figure 4.15: Comparison between analysis and simulation of Enabling-Disabling Interrupt model for system throughput

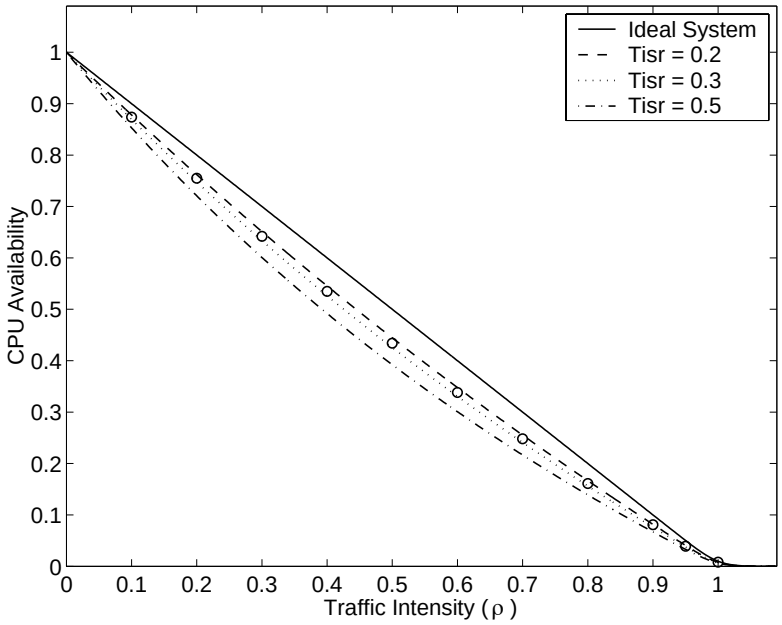


Figure 4.16: Comparison between analysis and simulation of Enabling-Disabling Interrupt model for CPU availability

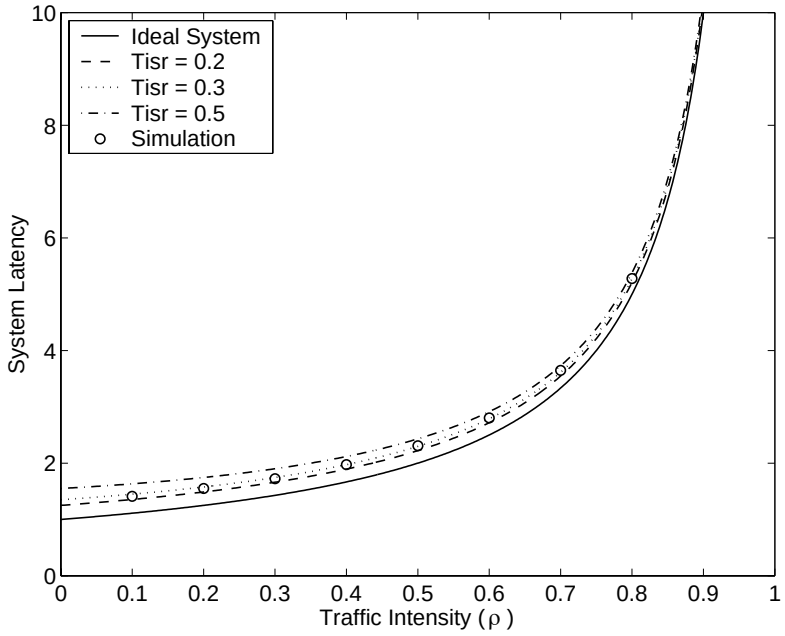


Figure 4.17: Comparison between analysis and simulation of Enabling-Disabling Interrupt model for system latency

CHAPTER 5

PERFORMANCE COMPARISON, DESGIN AND IMPLEMENTATION ISSUES

In this chapter, we present performance comparison of interrupt handling schemes using overall system power metric. The comparison depends on the design goal of the system where as the design goal depends on the weight of the system performance metrics which include throughput, latency, and CPU availability.

Some applications of computer networks, e.g. file transfer and video streaming, are throughput sensitive. The latency is generally not so important. Therefore, when we design a system for these applications, we give throughput more weight than latency and CPU availability. Other applications, e.g. voice over IP and interactive media, latency is more important than throughput. Therefore, latency is given more weight that throughput and CPU availability. When system responsiveness is concern to avoid user applications starvation, then we give more weight for CPU availability than throughput and latency. Equation (3-6) represents the overall system equation. The tunable parameters *a*, *b*, and *c* define the weights for throughput, CPU availability, and latency, respectively.

We compare the system performance of interrupt handling schemes for different design goal. The purpose is to find out which scheme is most suitable for this particular design goal. We start our comparison when we give equal weights for all goals. Then,

we evaluate the performance of interrupt handling schemes for different design goals. In all our comparisons, we fix the following system parameters: $T_{ISR} = 0.3$, $\mu = 1$, $\tau = 2$, and $T_{INT} = 0.05$.

5.1 Performance Compared

Design Example I. The goal of this design is to give equal weight for all system performance metrics. Figure 5.1 depicts the performance of interrupt handling schemes for this example where *a*, *b*, and *c* are equal to 1. We notice that Enabling-Disabling Interrupt scheme gives better performance. However, at low system load when $\rho < 0.1$, the performance of Traditional scheme is almost equivalent to the performance of Enabling-Disabling Interrupt scheme. We also notice that Traditional scheme gives better performance than Interrupt Coalescing scheme when traffic intensity is less than 0.4. In other words, Interrupt Coalescing gives more power than Traditional scheme at high arrival rate. Moreover, the optimal operating points for all schemes occur at low traffic intensity.

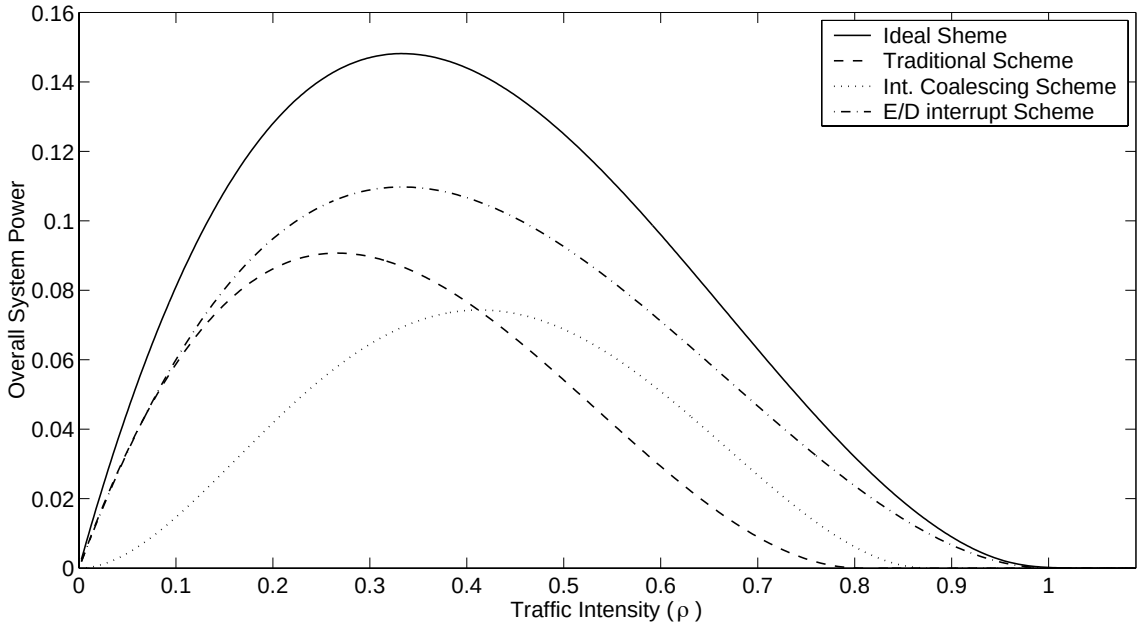


Figure 5.1: Performance of interrupt handling schemes

where all design goals have equal weights

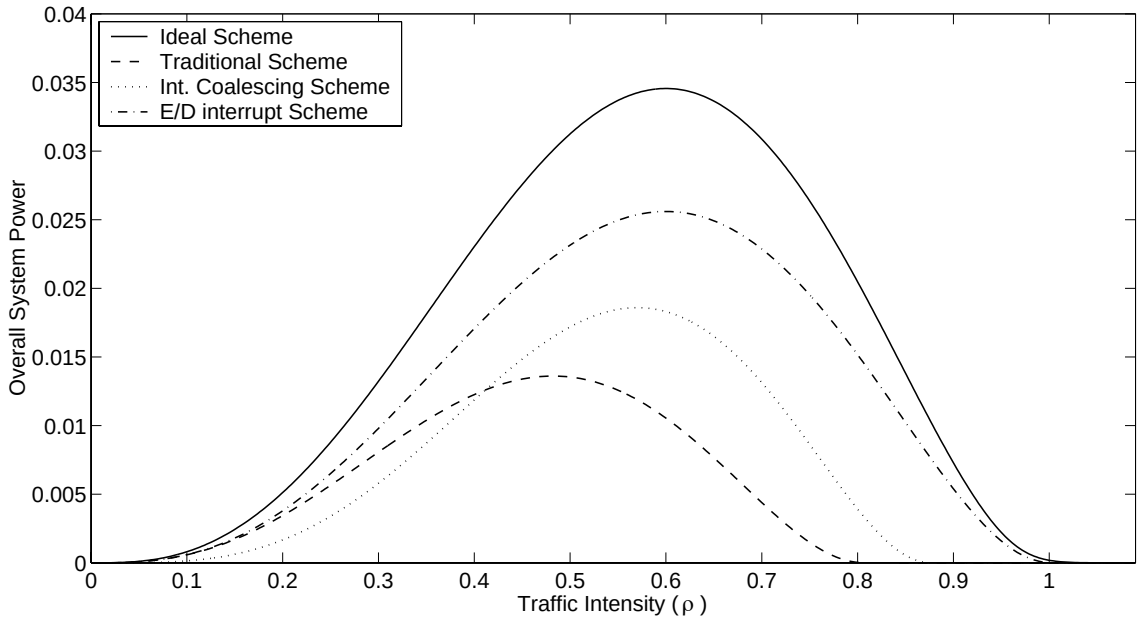


Figure 5.2: Performance of interrupt handling schemes where

system throughput has more weight than latency and CPU availability

Design Example II. The goal of this design is to give system throughput more weight than latency and CPU availability. Figure 5.2 illustrates the performance of interrupt handling schemes when $a = 3$, $b = 1$, and $c = 1$. We notice that Enabling-Disabling Interrupt scheme gives the best performance. We also notice that Traditional scheme gives better performance than Interrupt Coalescing scheme when traffic intensity is less than 0.4. However, the optimal operating points for all schemes occur at high traffic intensity.

Design Example III. The goal of this design is to give system latency more weight than throughput and CPU availability. Figure 5.3 shows the performance of interrupt handling schemes when $a = 1$, $b = 1$, and $c = 5$. It is noted that Traditional scheme give better performance at lower traffic intensity (i.e. $\rho < 0.1$). After this point, Enabling-Disabling Interrupt scheme gives more power than other schemes. We also notice that Interrupt Coalescing scheme is totally diminished. Moreover, we observe remarkable power degradation of interrupt handler schemes if we compare them with Ideal system.

Design Example IV. The goal of this design is to give CPU availability more weight than throughput and latency. Figure 5.4 illustrates the performance of interrupt handling schemes when $a = 1$, $b = 3$, and $c = 1$. We notice that Traditional scheme and Enabling-Disabling Interrupt scheme give approximately an equivalent power at lower traffic intensity. After this point, Enabling-Disabling Interrupt scheme has the best overall system power. It is noted that the power of Interrupt Coalescing scheme approaches the power of Enabling-Disabling scheme for higher traffic intensity.

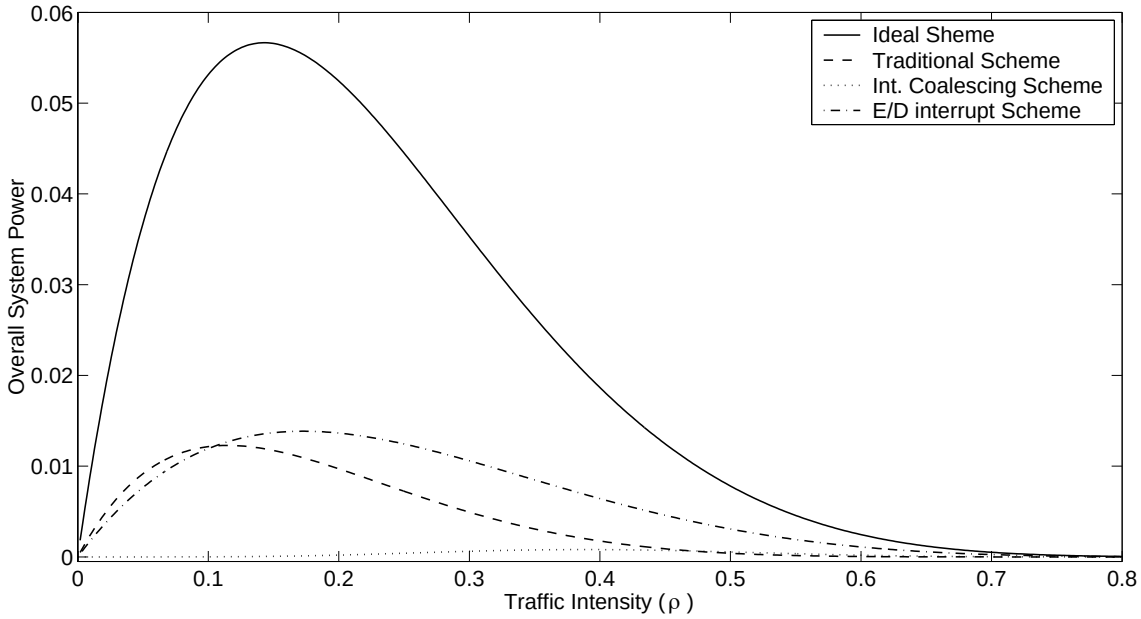


Figure 5.3: Performance of interrupt handling schemes where system latency has more weight than throughput and CPU availability

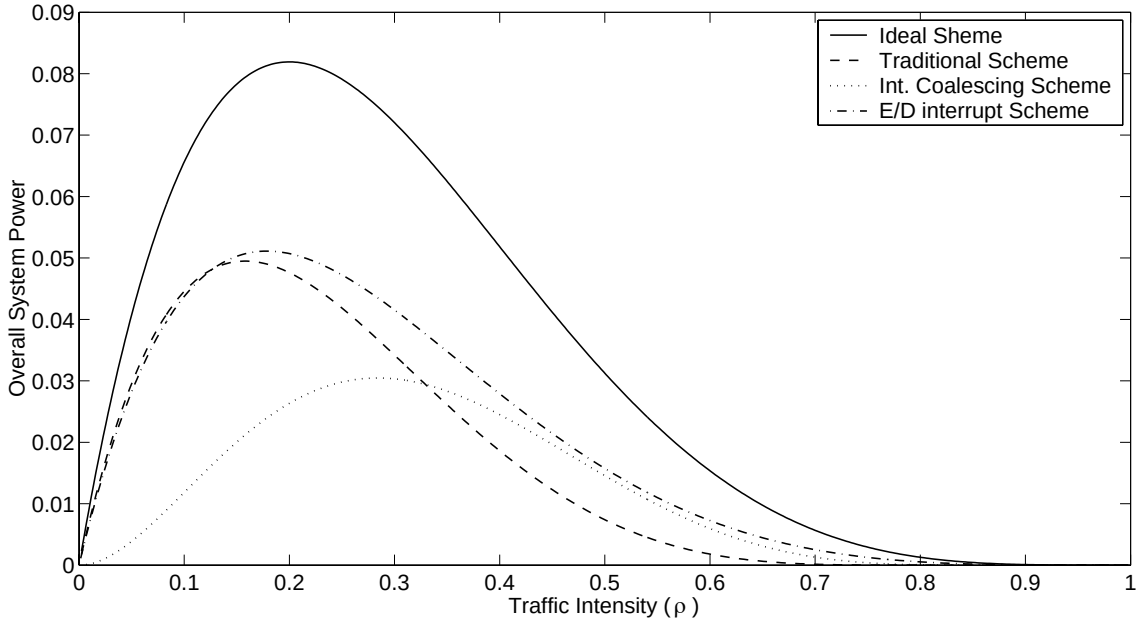


Figure 5.4: Performance of interrupt handling schemes where CPU availability has more weight than throughput and latency

Design Example V. The goal of this design is to give system throughput less weight than latency and CPU availability. Both system latency and CPU availability have similar weights. Figure 5.5 depicts the performance of interrupt handling schemes when $a = 0.5$, $b = 1$, and $c = 1$. We notice that Traditional scheme and Enabling-Disabling Interrupt scheme give approximately an equivalent power at lower traffic intensity. After this point, Enabling-Disabling Interrupt scheme has the best overall system power. Moreover, Interrupt Coalescing scheme outperforms Traditional scheme when traffic intensity is greater than 0.4.

Design Example VI. The goal of this design is to give system latency less weight than throughput and CPU availability. Both system throughput and CPU availability have similar weights. Figure 5.6 depicts the performance of interrupt handling schemes when $a = 1$, $b = 1$, and $c = 0.2$. We notice that all schemes give approximately an equivalent power at lower traffic intensity i.e., when $\rho < 0.2$. After this point, Interrupt Coalescing scheme has the best overall system power up to point when traffic intensity is less than 0.6. When traffic intensity is greater than 0.7, Enabling-Disabling Interrupt scheme gives better performance than other schemes.

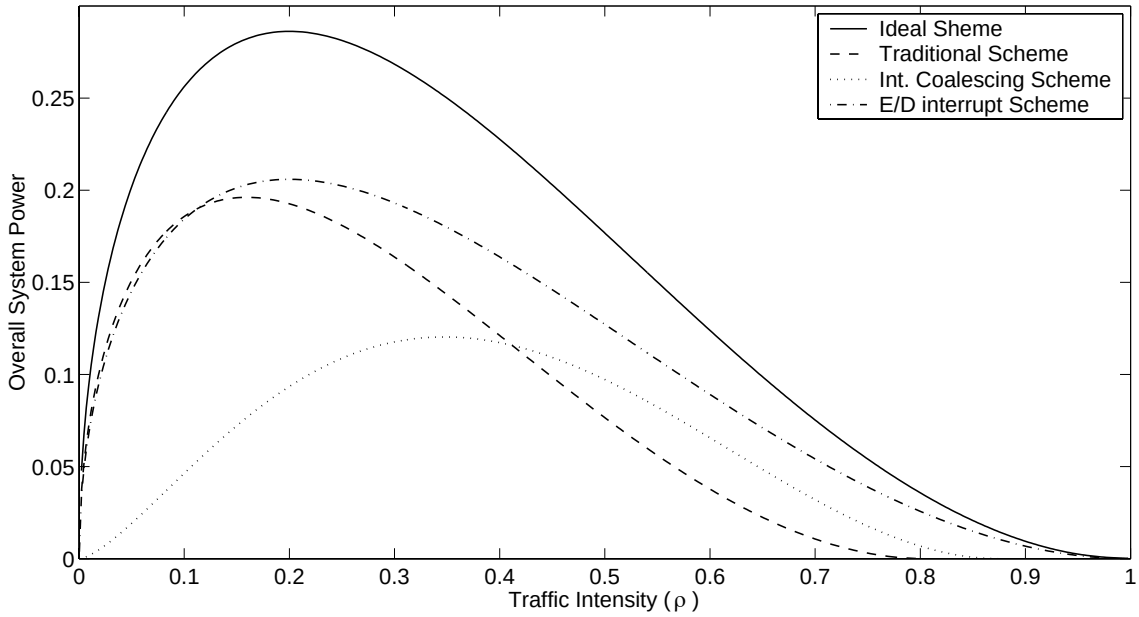


Figure 5.5: Performance of interrupt handling schemes where system throughput has less weight than latency and CPU availability

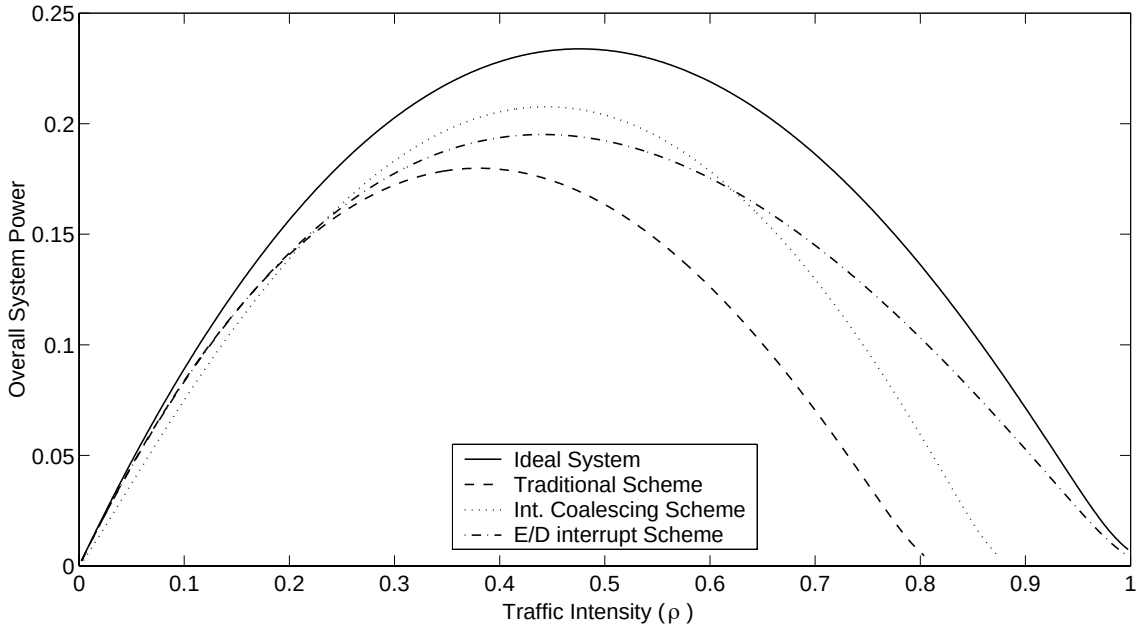


Figure 5.6: Performance of interrupt handling schemes where system latency has less weight than throughput and CPU availability

5.2 Selecting the Best Scheme

In this section, we will discuss the selection of proper scheme or schemes according to design goal. We see that Enabling-Disabling Interrupt scheme is a suitable scheme when the system goal is maximizing throughput, maximizing CPU availability, and minimizing latency. If the system does not support Enabling-Disabling Interrupt scheme, then we have to implement two schemes: Traditional and Interrupt Coalescing schemes (see Figure 5.1). The transition of these two schemes depends on the value of traffic intensity. If $\rho < 0.4$, then we switch to Traditional scheme. Otherwise, we switch to Interrupt Coalescing scheme.

If the system is sensitive to throughput then Enabling-Disabling Interrupt scheme outperforms Traditional and Interrupt Coalescing schemes (see Figure 5.2). If system responsiveness is to be considered also besides system throughput then we need to implement all these schemes (see Figure 5.6). The transition will depend on the system load. If $\rho < 0.2$ then we switch to Traditional scheme. If $0.2 < \rho < 0.65$ then we switch to Interrupt Coalescing scheme. Otherwise, we switch to Enabling-Disabling Interrupt scheme.

Finally, if the system is sensitive to latency then applying Interrupt Coalescing is not recommended. Traditional scheme is used when $\rho < 0.1$. Otherwise, Enabling-Disabling Interrupt scheme is used.

5.3 Design and Implementation Issues

We have seen that scheme's selection depends on system load. The system load is measured as traffic intensity ρ, where $\rho = \lambda / \mu$.

5.3.1 NIC-Side Solution

One way to estimate traffic intensity is to estimate packet arrival rate and packet processing time in protocol stack. We adopt the idea of [DOV01] to measure average packet arrival rate using exponential weighted average. The average packet interarrival $\hat{A}$ is estimated after each packet arrival using the following formula:

$$\hat{A} = \alpha \hat{A} + (1-\alpha)D \qquad (0 \le \alpha \le 1), \tag{5-1}$$

where α is the average interarrival weight factor and D is the duration between last packet arrival and the one before that. α controls the importance that is given to the last interarrival relative to the past history of interarrivals, as that is accumulated in $\hat{A}$.

The average processing time for a packet in protocol stack $\hat{S}$ can be estimated experimentally. The experiment runs an application that implements *loopback* interface. The application generates traffic and gathers two specific times. The first time is the time a packet has been sent to the interface. The second time is the receiving time for that packet. Then, the service time for this packet is half the difference between the two times. Therefore, we can compute the average service time $\hat{S}$.

The solution can be implemented in NIC provided that NIC can be programmed. Therefore, this solution will not produce any overhead inside the system kernel. The drawbacks of this technique are as follows. The accuracy of estimating interarrival time depends on α. We have to measure μ for each server we need to implement this technique.

5.3.2 OS-Side Solution

Another way to estimate traffic intensity is by estimating average number of packets in the host system memory $\hat{N}$ and probability of packet loss $\hat{p}_B$. Then, we can apply M/M/1/B model to compute $\hat{\rho}$ as follows:

$$\hat{N} = \frac{\hat{\rho}}{1-\hat{\rho}}\left[1-(B+1)\,\hat{p}_B\right],$$

$$\hat{\rho} = \frac{\hat{N}\left[1-(B+1)\,\hat{p}_B\right]}{1+\hat{N}\left[1-(B+1)\,\hat{p}_B\right]}. \qquad (5\text{-}2)$$

where $\hat{N} = \left(\sum N(t)\right)/n$ and $\hat{p}_B = \frac{(\text{Number of packets dropped})}{(\text{Total number of packets received})}$. n denotes the number of times we observe $N(t)$.

Initially, $\hat{N}$ and $\hat{p}_B$ are set to zero. Then, we update their values periodically, i.e., after a time slice of size T. The time T must be set in milliseconds (say 10 millisecond) in order to prevent producing overhead due to calculating $\hat{N}$ and $\hat{p}_B$.

However, setting T with bigger values may produce inaccurate prediction for $\hat{N}$ and $\hat{p}_B$.

This solution can be implemented inside the kernel. The disadvantage is the difficulty to set a suitable value for T to compromise between overhead and accuracy.

CHAPTER 6

CONCLUSION

This chapter presents a summary of our major contributions in this thesis work to study the operating system performance for different interrupt handling schemes. It also gives indications of future research directions.

One of our major contributions in this thesis is proposing the overall system power metric. This novel metric integrates three main metrics that measure the host system performance. The three metrics are system throughput, system latency and CPU availability for user processes.

We presented analytical models for interrupt handling schemes including Traditional scheme, Interrupt Coalescing scheme and Enabling-Disabling Interrupt scheme. First, we presented an analysis for the ideal situation in which the overhead involved in generating interrupts is totally ignored. Then, we presented two models for the Traditional scheme. The first model is based on analysis of the effective service rate. The second model uses pure Markovian model. The comparison results between the first and the second models are quit similar. Next, we modeled the Interrupt Coalescing scheme using analysis of the effective service rate. Finally, we modeled Enabling-Disabling Interrupt scheme using pure Markovian model.

We developed a simulation model to verify our analysis. Simulation results show that our analytical models are correct and accurate. We also verified our analysis by solving equations for special cases when the interrupt handling is ignored.

Performance comparison between interrupt handling schemes has been presented. We have shown that achieving the optimal system performance may require to implement different schemes for interrupt handling depending on the current system load. We also discussed some implementations issues related to estimating the system load.

The topics presented in this thesis open a new horizon for further research. The followings are some future directions:

- In our analysis, we assumed that all packets have fixed size length. Further analysis is needed to consider general distribution for packet length. Simulation model is also needed to verify the analysis. Such analysis and simulation can help studying system performance of Gigabit Ethernet jumpo frames.
- In this thesis, we used a Poisson process to model traffic source. In realistic settings, traffic sources are bursty. This behavior can be modeled using Pareto distribution [LEL94]. Therefore, further work is needed to examine system performance using Pareto distribution.
- Interrupt handling schemes described in this thesis run at full speed, i.e., protocol stack routine will keep processing as long as there is a packet in the kernel memory. This will cause starvation problem at high load. One

solution is to implement polling[1]. In polling, packet quota is used to limit the number of packets to be processed in each poll. This will prevent protocol stack processing to consume all CPU resources. Further work is needed to model, analyze, and simulate polling scheme.

- Prototype or experimental implementation is needed to validate analytical and simulation results for the different interrupt handling schemes. A typical prototype or experiment would involve two PCs equipped with GbE NICs running Linux OS. Modifications to the OS kernel and device drivers would be required.

[1] Look at section 2.2.3 for more details.

Appendix A

M/M/1 Queue

1. Traffic intensity: $\rho = \lambda / \mu$.
2. Stability condition: Traffic intensity ρ must be less than 1.
3. Probability of zero packets in the system: $p_0 = 1 - \rho$.
4. Probability of *n* packet in the system: $p_n = (1-\rho)\rho^n$, $n = 0, 1, \ldots, \infty$.
5. Average number of packet in the system: $E(n) = \rho/(1-\rho)$.
6. Mean response time: $R = (1/\mu)/(1-\rho)$.

M/M/1/B Queue

1. Traffic intensity: $\rho = \lambda / \mu$.
2. The system is always stable: $\rho < \infty$.
3. Probability of zero packets in the system:

$$p_0 = \begin{cases} \dfrac{1-\rho}{1-\rho^{B+1}} & \rho \neq 1 \\ \dfrac{1}{B+1} & \rho = 1 \end{cases}.$$

4. Probability of *n* packet in the system:

$$p_n = \begin{cases} \dfrac{1-\rho}{1-\rho^{B+1}}\rho^n & \rho \neq 1 \\ & 0 \le n \le B \\ \dfrac{1}{B+1} & \rho = 1 \\ 0 & n > B \end{cases}.$$

5. Average number of packet in the system:

$$E(n) = \frac{\rho}{1-\rho} - \frac{(B+1)\rho^{B+1}}{1-\rho^{B+1}}.$$

6. Mean response time:

$$R = \frac{E(n)}{\lambda(1-p_B)}.$$

Appendix B

Simulation Code

```
#include "rand.h"
#include "mm1b.h"

int main(int argc, char *argv[])
{
      event_type next_event;
      double lower_rate, upper_rate, service_rate, arrival_rate;
      uint num_of_events;
      int model;

      printf("1- Normal interrupt model (case 1)\n");
      printf("2- Normal interrupt model (case 2)\n");
      printf("3- Interrupt coalescing model\n");
      printf("4- Enabling/Disabling interrupt model\n");
      printf("select --> "); scanf("%d", &model);

      /* collect data from user */
      printf("Enter arrival range [lower rate, upper rate] : ");
      scanf("%lf %lf", &lower_rate, &upper_rate);
      printf("Enter service rate: ");
      scanf("%lf", &service_rate);
      printf("Enter ISR time : ");
      scanf("%lf", &mean_isr);
      if (model == 3) {
            printf("Enter coalescing size : ");
            scanf("%d", &coal_size);
      }
      if (model == 4)
            time_instructions = 0.05;
      else
            time_instructions = 0;
      printf("Enter system buffer size [0 for infinity]: ");
      scanf("%ld", &buf_size);
      while (buf_size > MAX_BUFFER_SIZE) {
            printf("Error: You buffer size exceed the maximum limit
%lu\n", MAX_BUFFER_SIZE);
            printf("Enter system buffer size [0 for infinity]: ");
            scanf("%ld", &buf_size);
      }
      model_type = (buf_size == 0)? MM1,buf_size=MAX_BUFFER_SIZE:MM1B;

      /* set system parameters */
      mean_service      = 1.0 / (service_rate);  /* 1 / rate */
      printf("\n<<<<<<<<<<<<  SYSTEM PARAMETERS >>>>>>>>>>>>>>\n");
      printf("Simulation model # .............  = %d\n", model);
      printf("Mean service rate...............  = %6.4f\n",
1.0/mean_service);
      printf("Mean ISR .......................  = %6.4f\n", mean_isr);
      if (model == 3)
            printf("Coalescing size ..........  = %d\n", coal_size);
      printf("Buffer size.....................  = %lu\n", buf_size);
      printf("-----------------------------------------------\n\n");
      printf(" lamda  P(B)     rho   CPU_PKT  CPU_ISR    Th   E(r)\n");
```

```
	arrival_rate = lower_rate;
	while ( arrival_rate <= upper_rate) {
		mean_interarrival = 1.0 / (arrival_rate);	/* 1 / rate */

		/* Initialize the simulation model */
		num_of_events = 0;
		initialize();

		while (num_of_events++ < 800000) {
			/* determine the next event */
			next_event = timing();

			/* Update time-average statistical accumulators */
			update_time_avg_stats();

			/* invoke the appropriate event function */
			switch (next_event) {
			case ARRIVAL:
				switch (model) {
				case 1: normal_arrive_1(); break;
				case 2: normal_arrive_2(); break;
				case 3: coal_arrive(); break;
				case 4: ed_arrive(); break;
				}
				break;
			case DEPARTURE: depart(); break;
			case ISR: interupt(); break;
			}
		}
		report();
		arrival_rate += 0.1;
	}
	return 0;
}

/*************************************************************/
/*
/*		initialize routine
/*
/*************************************************************/
void initialize(void)
{
	/* Initialize the simulation clock */

	time = 0.0;

	/* Initialize the state variables */

	protocol_status		= IDLE;
	isr_status		= IDLE;
	num_in_sys		= 0L;
	time_last_event		= 0.0;
	interrupt_enabled		= TRUE;
	rem_packets		= coal_size;
```

```
      /* Initialize the statistical counters */

      area_num_in_q       = 0.0;
      area_num_in_sys     = 0.0;
      area_server_status  = 0.0;
      num_pack_depart     = 0;
      total_response_time = 0.0;
      num_arrival         = 0L;
      num_pack_drop       = 0L;
      area_cpu_protocol   = 0.0;
      area_cpu_isr        = 0.0;

      /* Initialize event list.  Since no packets are present, the
      departure (service completion) event is eliminated from
      consideration.
      */

      time_next_event[ARRIVAL] = time +
expon_arrival(mean_interarrival);
      time_next_event[DEPARTURE] = INFINITY;
      time_next_event[ISR] = INFINITY;

      /* initialize buffer */
      head = tail = 0;
}
/*****************************************************************/
/*
/*                timing routine
/*
/*****************************************************************/
event_type timing(void)
{
      double min_time;
      event_type event;

      /* since we always schedule next arrival whenever packet arrives
      we assume this event will happen next unless other events occur
      before next packet arrival
      */
      min_time = time_next_event[ARRIVAL]; /* since we always expect an
arrival packet */
      event = ARRIVAL;

      /* now check if other events occur before arrival event, with
following consideration:
      1- if server execute an ISR, departure event will not be
scheduled next.
      2- otherwise, select either arrival event or departure event.
      */
      if (isr_status == TRUE) /* if the server executes an ISR */
      {
            if (time_next_event[ISR] <= min_time) {
                  min_time = time_next_event[ISR];
```

```
                event = ISR;
            }
        }
        else /* no isr */
        {
            if (time_next_event[DEPARTURE] < min_time) {
                min_time = time_next_event[DEPARTURE];
                event = DEPARTURE;
            }
        }

        /* advance the simulation clock */
        time = min_time;

        return event;
}

/*****************************************************************/
/*
/*                Arrival event routines
/*
/*****************************************************************/
/*****************************
* Normal interrupt model case 1
*****************************/
void normal_arrive_1(void)
{
        /* schedule next arrival */
        time_next_event[ARRIVAL] = time +
expon_arrival(mean_interarrival);

        /* incement number of packet arrival */
        num_arrival++;

        /* if packet arrived ouside an isr, then generate an interrupt */
        if (isr_status == IDLE) {
            time_isr = expon_isr(mean_isr) + time_instructions;
            time_next_event[ISR] = time + time_isr; /* time where isr
finish its execution */
            isr_status = BUSY;
        }

        /* Check to see if there is place in the buffer */
        if (num_in_sys < buf_size) {
            num_in_sys++;
            queue[tail] = time;
            tail  = ++tail % buf_size;
        }
        else /* drop the packet */
            if (model_type == MM1) {
                printf("\nInsufficient memory ....");
                exit(0);
            }
            else num_pack_drop++;
```

```
}

/*****************************
* Normal interrupt model case 2
*****************************/
void normal_arrive_2(void)
{
      /* schedule next arrival */
      time_next_event[ARRIVAL] = time +
expon_departure(mean_interarrival);

      /* incement number of packet arrival */
      num_arrival++;

      /* Check to see if there is place in the buffer */
      if (num_in_sys < buf_size) {
            num_in_sys++;
            queue[tail] = time;
            tail  = ++tail % buf_size;
            /* if packet arrived ouside an isr, then generate an
interrupt */
            if (isr_status == IDLE) {
                  time_isr = expon_isr(mean_isr) + time_instructions;
                  time_next_event[ISR] = time + time_isr; /* time where
isr finish its execution */
                  isr_status = BUSY;
            }
      }
      else /* drop the packet */
            num_pack_drop++;
}

/*********************
* Interrupt coalescing model
*********************/
void coal_arrive(void)
{
      /* schedule next arrival */
      time_next_event[ARRIVAL] = time +
expon_arrival(mean_interarrival);

      /* incement number of packet arrival */
      num_arrival++;

      /* check if we have to generate an interrupt or not */
      if (--rem_packets <= 0) {
            /* remask interrupt coalescing */
            rem_packets = coal_size;
            /* if packet arrived ouside an isr, then generate an
interrupt */
            if (isr_status == IDLE) {
                  time_isr = expon_isr(mean_isr);
                  time_next_event[ISR] = time + time_isr; /* time where
isr finish its execution */
```

```
                  isr_status = BUSY;
            }
      }
      /* Check to see if there is place in the buffer */
      if (num_in_sys < buf_size) {
            num_in_sys++;
            queue[tail] = time;
            tail  = ++tail % buf_size;
      }
      else /* drop the packet */
            if (model_type == MM1) {
                  printf("\nInsufficient memory ....");
                  exit(0);
            }
            else num_pack_drop++;
}

/**********************
* Enabling/Disabling interrupt model
**********************/
void ed_arrive(void)
{
      /* schedule next arrival */
      time_next_event[ARRIVAL] = time +
expon_arrival(mean_interarrival);

      /* incement number of packet arrival */
      num_arrival++;

      /* check if interrupt is enabled or not */
      if (interrupt_enabled) {
            interrupt_enabled = FALSE;
            /* if packet arrived ouside an isr, then generate an
interrupt */
            time_isr = expon_isr(mean_isr);
            time_next_event[ISR] = time + time_isr; /* time where isr
finish its execution */
            isr_status = BUSY;
      }

      /* Check to see if there is place in the buffer */
      if (num_in_sys < buf_size) {
            num_in_sys++;
            queue[tail] = time;
            tail  = ++tail % buf_size;
      }
      else /* drop the packet */
            if (model_type == MM1) {
                  printf("\nInsufficient memory ....");
                  exit(0);
            }
            else num_pack_drop++;
}
```

```
/*****************************************************************/
/*
/*                Departure event routine
/*
/*****************************************************************/
void depart(void)
{
      double response_time;

      num_in_sys--;

      /* compute the response time of the depart's packet and update
the
      total response time */

      response_time = time - queue[head]; /* total time elapsed in the
system */
      total_response_time += response_time;

      /* remove the served packet */
      head = ++head % buf_size;

      /* increment the number of packets departed */
      num_pack_depart++;

      /* check wether the queue is empty */
      if (num_in_sys <= 0) {
    /* no packet in the system. so make the server IDLE and eliminate
the
            departure event from consideration */
            protocol_status = IDLE;
            interrupt_enabled = TRUE;                              /*
this is for enabling/disabling scheme */
            time_next_event[DEPARTURE] = INFINITY;
      }
      else
            /* if queue is not empty, schedule the next packet */
            time_next_event[DEPARTURE] = time +
expon_departure(mean_service);
}

/*****************************************************************/
/*
/*                ISR event routine
/*
/*****************************************************************/
void interupt(void)
{
      /* finishing ISR */
      isr_status = IDLE;
      time_next_event[ISR] = INFINITY;

      /* check if the server was busy or not */
      if (protocol_status == BUSY)
```

```
            /* delay the current packet departure time */
            time_next_event[DEPARTURE] += time_isr;
      else {
          /* if server was IDLE, then schedule the departure time for
incoming
             packet and set the server to BUSY */
            time_next_event[DEPARTURE] = time +
expon_departure(mean_service);
            protocol_status = BUSY;
      }
}

/*************************************************************/
/*
/*                Update area accumulators routine
/*
/*************************************************************/
void update_time_avg_stats(void)
{
      double time_since_last_event;

      /* compute time since last event, and update last-event-time
      marker */
      time_since_last_event = time - time_last_event;
      time_last_event = time;

      /* update all area */
      area_num_in_sys += num_in_sys * time_since_last_event;

      /* update server utilization, since server could be busy with isr
or
      packet processing */
      area_server_status += (protocol_status | isr_status) *
time_since_last_event;
      area_cpu_protocol += (~isr_status & protocol_status) *
time_since_last_event;
      area_cpu_isr += isr_status * time_since_last_event;
}

/*************************************************************/
/*
/*                exponential variate generation routine
/*
/*************************************************************/
double expon_departure (double mean)
{
      double u;

      /* Generate a U(0,1) random variate */

      u = lcg_rand(1);

      /* Return an exponential random variate with mean "mean" */
      return (-mean * log(u));
```

```
}

double expon_isr (double mean)
{
      double u;

      /* Generate a U(0,1) random variate */

      u = lcg_rand(50);

      /* Return an exponential random variate with mean "mean" */
      return (-mean * log(u));
}
double expon_arrival (double mean)
{
      double u;

      /* Generate a U(0,1) random variate */

      u = lcg_rand(99);

      /* Return an exponential random variate with mean "mean" */
      return (-mean * log(u));
}

/*************************************************************************/
/*
/*               Report routine
/*
/*************************************************************************/
void report(void)
{
      double average_num_in_sys;
      double throughput;
      double mean_response_time;
      double server_util;

      average_num_in_sys = area_num_in_sys / time;
      mean_response_time = total_response_time / num_pack_depart;
      throughput = num_pack_depart / time;
      server_util = area_server_status / time;

      printf("%6.4f\t%4.3f\t%4.3f\t%4.3f\t%4.3f\t%6.3f\t%12.3f\t%12.3f\
n", 1.0/mean_interarrival,
            (double)num_pack_drop/num_arrival, server_util,
area_cpu_protocol/time, area_cpu_isr/time,
            throughput, mean_response_time, average_num_in_sys);
}
```

```
#ifndef _MM1B_H_
#define _MM1B_H_

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/* mnemonics for event types */
typedef enum {ARRIVAL=0, DEPARTURE, ISR, NUM_EVENTS} event_type;
/* mnemonics for server's being idle or busy */
typedef enum {IDLE=0, BUSY} status;
/* mnemonics for boolean values */
enum {FALSE=0, TRUE};
enum {MM1=0, MM1B};

typedef unsigned long uint;

#define INFINITY   1E+20            /* infinity time */

/* System parameters */
double mean_interarrival;           /* mean interarrival time of
packets */
double mean_service;                /* mean service time of packets */
double mean_isr;                    /* mean interrupt service routine
time */
uint   buf_size;                    /* memory buffer size */

#define MAX_BUFFER_SIZE     1000000 /* Maximum buffer size */

/* System variables */
int         isr_status;                  /* Is server processing ISR
or not */
int         protocol_status;        /* Is server processing protocol
stack or not */
uint  num_in_sys;                   /* number of packets in the system
*/
double      queue[MAX_BUFFER_SIZE]; /* system buffer , it holds only
arrival time for each packet */
double      time;                        /* simulation clock time */
double      time_last_event;        /* time of last event */
double      time_isr;                    /* time of isr handling */
double      time_next_event[NUM_EVENTS];   /* next event list */
int         coal_size;                   /* number of packets to be
coalesced */
int         rem_packets;                 /* number of packets remained
to generate an isr */
int         model_type;                  /* is M/M/1 or M/M/1/B */
int         interrupt_enabled;           /* interrupt is enabled or
disabled */
double      time_instructions;           /* time to execute two
instructions for enabling/disabling interrupts */

/* Statistic variables */
```

```
uint   num_pack_depart;       /* number of packets leave the system
successfully */
double area_num_in_q;
double area_num_in_sys;
double area_server_status;
double total_response_time;
uint   num_pack_drop;
uint   num_arrival;
double area_cpu_protocol;
double area_cpu_isr;

/* buffer manipulation */
uint head;
uint tail;

/* function prototypes */
void initialize(void);
event_type timing (void);
void normal_arrive_1(void);
void normal_arrive_2(void);
void coal_arrive(void);
void ed_arrive(void);
void depart(void);
void interupt(void);
void update_time_avg_stats(void);
double expon_departure(double);
double expon_arrival(double);
double expon_isr(double);
void report(void);

#endif
```

```
#include "rand.h"
/* Prime modulus multiplicative Linear Congruential Generator (LCG)
   Z[i] = (630360016 * Z[i-1]) (mod(pow(2,31) - 1)), based on Marse and
   Roberts' portable FORTRAN random-number generator UNIRAN.  Multiple
   (100) streams are supported, with seeds spaced 100,000 apart.
   Throughout, input argument "stream" must be an int giving the
   desired stram number.  The header file rand.h must be included
   in the calling program before using these functions.

   Usage:

     1. To obtain the next U(0,1) random number from stram "stream"
        execute:
             u = lcg_rand(stram);
        where lcg_rand is a float function.  The float variable u will
        contain the next random number.

     2. To set the seed for stream "stream" to a desired value zset,
        execute
             randst(zset, stream);
        where randst is a void function and zset mub be a long set to
        desired seed, a number between 1 and 2147483646 (inclusive).
        Default seeds for all 100 streams are given in the code.

     3. To get the current (most recently used) integer in the sequence
        being generated for stram "stream" into the long variable zget,
        execute
             zget = randgt(stream);
        where randgt is a long function

*/


/* Define the constants. */

#define MODLUS 2147483647
#define MULT1       24112
#define MULT2       26143



/* Set the default seeds for all 100 streams */

static long zrng[]=
{         0,
 1973272912, 281629770,  20006270,1280689831,2096730329,1933576050,
  913566091, 246780520,1363774876, 604901985,1511192140,1259851944,
  824064364, 150493248, 242708531,  75253171,1964472944,1202299975,
  233217322,1911216000, 726370533, 403498145, 993232223,1103205531,
  762430696,1922803170,1385516923,  76271663, 413682397, 726466604,
  336157058,1432650381,1120463904, 595778810, 877722890,1046574445,
   68911991,2088367019, 748545416, 622401386,2122378830, 640690903,
 1774806513,2132545692,2079249579,  78130110, 852776735,1187867272,
 1351423507,1645973084,1997049139, 922510944,2045512870, 898585771,
```

```
  243649545,1004818771, 773686062, 403188473, 372279877,1901633463,
  498067494,2087759558, 493157915, 597104727,1530940798,1814496276,
  536444882,1663153658, 855503735,  67784357,1432404475, 619691088,
  119025595, 880802310, 176192644,1116780070, 277854671,1366580350,
 1142483975,2026948561,1053920743, 786262391,1792203830,1494667770,
 1923011392,1433700034,1244184613,1147297105, 539712780,1545929719,
  190641742,1645390429, 264907697, 620389253,1502074852, 927711160,
  364849192,2049576050, 638580085, 547070247  };


/* Generate the next random number. */

double lcg_rand(int stream)
{
   long zi, lowprd, hi31;

   zi     = zrng[stream];
   lowprd = (zi & 65535) * MULT1;
   hi31   = (zi >> 16) * MULT1 + (lowprd >> 16);
   zi     = ((lowprd & 65535) - MODLUS) +
            ((hi31 & 32767)  << 16)  + (hi31 >> 15);
   if (zi < 0) zi += MODLUS;
   lowprd = (zi & 65535) * MULT2;
   hi31   = (zi >> 16) * MULT2 + (lowprd >> 16);
   zi     = ((lowprd & 65535) - MODLUS) +
            ((hi31 & 32767)  << 16)  + (hi31 >> 15);
   if (zi < 0) zi += MODLUS;
   zrng[stream] = zi;
   return ((zi >> 7 | 1) + 1)/ 16777216.0;
}

/* Set the current zrng for stream "stream" to zset. */

void randst (long zset, int stream)
{
   zrng[stream] = zset;
}

/* Return the current zrng for stream "stream". */

long randgt (int stream)
{
   return zrng[stream];
}
```

```
/*
      The following declarations are for use of the random-number
   generator rand and the associated functions randst and randgt for
   seed management.  This file (named rand.h) should be included
   in any program using these functions as follows:
*/

#ifndef _RAND_H_
#define _RAND_H_

double lcg_rand(int stream);
void randst(long zset, int stream);
long randgt(int stream);

#endif
```

Bibliography

[ALTE] Alteon WebSystems Inc, *Jumbo Frames*, www.alteon-websystems.com /products/white_papers/jumbo.

[ARON99] Aron, M. and Drushel, P., Soft Timers: Efficient Microsecond Software Timer Support for Network Processing, *In Proceeding of the 17th Symp. on Operating systems Principles*, pages 232-246, Kiawah Island Resort, SC, December 1999.

[ARON00] Aron, M. and Drushel, P., Soft Timers: Efficient Microsecond Software Timer Support for Network Processing, *ACM Transactions on Computer Systems*, vol. 18, pp. 197-228, Aug. 2000.

[BHO00] Bhoedjang, R., Verstoep, K., Ruhl, T., Bal, H., and Hofman, R., Evaluating Design Alternatives For Reliable Communication On High-Speed Networks. *In Proceedings of ASPLOS-9*, November 2000.

[BOD95] Boden, N., Cohen, D., and Felderman, R., Myrinet: A Gigabit Per Second Local-Area Network, *IEEE Micro*, 15(1):29, February 1995.

[CERN] *CERN AceNIC Linux Driver*, http://jes.home.cern.ch/jes/gige/acenic.html

[DAY83] Day, J. D., and Zimmerman, H., The OSI Reference Model. *Proceedings of the IEEE*, vol. 71, pp. 1334-1340, 1983.

[DEC] *21143 PCI Lan Controller Hardware* Reference Manual. http://www.intel.com/design/network/manuals/278074.htm.

[DOV01] Dovrolis, C., Thayer, A., and Ramanathan, P., HIP: Hybrid Interrupt-Pollling for the Network Interface, *ACM OS Reviews*, vol 35, pp. 50-60, Oct. 2001.

[ELAY96] Elaydi, S. N., *An Introduction to Difference Equations*, Springer-Verlag 1996, pg 113.

[EICK95] Eicken, T., A. Basu, V. Buch, and W. Vogels, U-Net: A User-Level Network Interface For Parallel And Distributed Computing. *In Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[FAR00] Farrel, P. and Ong, H., Communication Performance over a Gigabit Ethernet Network, *IEEE Proc. of 19th IPCCC*, 2000.

[GIES78] Giessler, A., Haanle, J., Konig, A., and Pade, E., Free Buffer Allocation – An Investigation by Simulation, *Computer Networks*, vol. 1, no. 3, pp. 191-204, July 1978.

[GRO98] Gross, D. and Harris, C. M., *Fundamentals of Queueing Theory*. John Wiley & Sons. 3rd Edition, 1998.

[HAN97] Hansen, J. S. and Jul, E., A Scheduling Scheme for Network Saturated NT Multiprocessor, *In Proceeding of USENIX Window NT Workshop*, Seattle, Washington, August 1997.

[HAS00] Hasegawa, Y., Nagasaka, Y., and Yasu, Y., *DAQ/EF-1 Event Builder system on Linux/Gigabit Ethernet*, http://rd13doc.cern.ch/Atlas/Notes/147/ Note147-1.html.

[IND98] Indiresan, A., Mehra, A., and Shin, K. G., Receive Livelock Elimination via Intelligent Interface Backoff, *TCL Technical Report*, University of Michigan, 1998.

[KIM01] Kim, I., Moon, J., and Yeom, H. Y., Timer-Based Interrupt Mitigation for High Performance Packet Processing, *5th International Conference on High-Performance Computing in the Asia-Pacific Region*, September, 2001, Gold Coast, Australia.

[KLEI93] Kleinroch, L., On the Modeling and Analysis of Computer Networks, *Proc. of IEEE*, vol. 81, no. 8, pp. 1179-1191, August 1993.

[JAIN88] Jain, R., Ramakrishnan, K. K., Congestion Avoidance in Computer Networks with a Connectionless Network Layer: Concepts, Goals and Methodology, *Proceedings of Computer Networking Symposium*, pp 134-143, 1988.

[JAIN90] Jain, R., Congestion control in Computer Networks: Issues and Trends, *IEEE Network Magazine*, May 1990.

[LAI96] Lai, K. and Baker M., A Performance Comparison Of Unix Operating Systems On The Pentium, *In The Proceedings of the Winter 1996 USENIX Technical Conference*, Jan. 1996

[LAW91] Law, A. M. and Kelton, W. D., *Simulation Modeling and Analysis*, McGraw-Hill. 2nd Edition, 1991.

[LEL94] Leland, W., Taqqu, M., Willinger, W., Wilson, D., On the Self-Similar Nature of Ethernet Traffic, *IEEE/ACM Trans. On Networking*, vol. 2, pp. 1-15, 1994.

[MAQ96] Maquelin, O., Gao, G. R., Hum, H. H., Theobald, K. B., and Tian, X., Polling Watchdog: Combining Polling and Interrupts for Efficient Message Handling, *In Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 179-190, New York, May 1996. ACM Press.

[MOG97] Mogul, J., and Ramakrishnan, K. K., Eliminating Receive Livelock In An Interrupt-Driven Kernel, *ACM Trans. Computer Systems*, vol. 15, no. 3, pp. 217-252, August 1997.

[PAK95] Pakin, S., Lauria, M., and Chien, A., High Performance Messaging On Workstations: Illinois Fast Messages (FM) For Myrinet, 1995.

[PIE01a] Pietikainen, P., Scheduled Transfer Protocol on Linux. 2001.

[PIE01b] Pietikainen, P., *Hardware-Assisted Networking Using Scheduled Transfer* Protocol on Linux. Ph.D thesis, 2001.

[RAM93] Ramakrishnan, K. K., Performance Considerations in Designing Network Interfaces, *IEEE Journal on Selected Areas in Communications* 11(2): 203-219.1993

[RIL00] Riley, S., Breyer, R., Switched, Fast, and Gigabit Ethernet, *Mtp Network Engineering Series*. 2000

[RIZZ02] Rizzo, L., *Device Polling support for FreeBSD*, http://info.iet.unipi.it/~luigi/polling/, online document, Feb. 2002.

[RUB01] Rubini, A. and Corbet, J., *Linux Device Drivers. 2nd Edition*. O'Reilly,2001

[PAR02] Parker, M., A Case for User-Level Interrupts. 2002

[PAT03] Patterson, D. A., and Hennessy, J. L., Computer Architecture, A Quatitative Approach. 3rd Edition. Morgan Kaufmann, 2003
.

[SHIV01] Shivan, P., Wyckoff, P., and Panda, D., EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet Message Passing, *In Proceedings of SC2001, Denver*, Colorado, USA, November 2001.

[SMI93] J. M. Smith and C. B. S. Traw., Giving applications access to Gb/s networking, *IEEE Network*, 7(4):44-52, July 1993.

[STE94] Steenkiste, P. A., A Systematic Approach to Host Interface Design for High-Speed Networks, *IEEE Computer*, 27(3):47-57, March 1994.

[SUM98] Sumimoto, S., Tezuka, H., Hori, A., Harada, H., Takahashi, T., and Ishikawa, Y., High Performance Communication Using A Gigabit Ethernet. *Technical Report TR-98003*, Real World Computing Partnership, 1998.

[TRI98] Trivedi, B., *Queueing Networks and Markov Chains*, John Wiley & Sons. 1998.

[VAH96] Vahalia, U., *UNIX Internals, the new fronties*, Prentice Hall, 1996.

[XU98] Xu, C., Han, X., Liu, C., and Mann, J., Active Messages Using Selective Interrupts Without Polling, *In Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Systems*, Oct. 1998, Las Vagas, NV.

Vita

- Khalid Abdalla El-Badawi.
- Born in Kuwait on November 11, 1970.
- Completed Bachelor of Science (B.Sc.) in Joint Subject of Mathematical and Computer Science from University of Khartoum, Sudan, in December 1994.
- Completed MS in Computer Science from King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia, in April 2003.
- Email: badawikhalid@hotmail.com.